Key Constraints and Monotonic Aggregates in Deductive Databases

Carlo Zaniolo

Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095
zaniolo@cs.ucla.edu
http://www.cs.ucla.edu/~zaniolo

Abstract. We extend the fixpoint and model-theoretic semantics of logic programs to include unique key constraints in derived relations. This extension increases the expressive power of Datalog programs, while preserving their declarative semantics and efficient implementation. The greater expressive power yields a simple characterization for the notion of set aggregates, including the identification of aggregates that are monotonic with respect to set containment and can thus be used in recursive logic programs. These new constructs are critical in many applications, and produce simple logic-based formulations for complex algorithms that were previously believed to be beyond the realm of declarative logic.

1 Introduction

The basic relational data model consists of a set of tables (or base relations) and of a query language, such as SQL or Datalog, from which new relations can be derived. Unique keys can be declared to enforce functional dependency constraints on base relations, and their important role in database schema design has been recognized for a long time [1,28]. However, little attention has been paid so far to the use of unique keys, or functional dependencies, in derived relations. This paper shows that keys in derived relations increase significantly the expressive power of the query languages used to define such relations and this additional power yields considerable benefits. In particular, it produces a formal treatment of database aggregates, including user-defined aggregates, and monotonic aggregates, which can be used without restrictions in recursive queries to express complex algorithms that were previously considered problematic for Datalog and SQL.

2 Keys on Derived Relations

For example, consider a database containing relations `student`(`Name`, `Major`), and `professor`(`Name`, `Major`). In fact, let us consider the following microcollege example that only has three facts:

A.C. Kakas, F. Sadri (Eds.): Computat. Logic (Kowalski Festschrift), LNAI 2408, pp. 109–134, 2002.

$$\texttt{student}(\texttt{'JimBlack'}, \texttt{ee}). \qquad \texttt{professor}(\texttt{ohm}, \texttt{ee}).$$
$$\texttt{professor}(\texttt{bell}, \texttt{ee}).$$

Now, the rule is that the major of a student must match his/her advisor's main area of specialization. Then, eligible advisors can be computed as follows:

$$\texttt{elig\_adv}(\texttt{S}, \texttt{P}) \leftarrow \texttt{student}(\texttt{S}, \texttt{Majr}),\ \texttt{professor}(\texttt{P}, \texttt{Majr}).$$

Now the answer to a query ?$\texttt{elig\_adv}(\texttt{S}, \texttt{P})$ is

$$\{\texttt{elig\_adv}(\texttt{'JimBlack'}, \texttt{ohm}),\ \texttt{elig\_adv}(\texttt{'JimBlack'}, \texttt{bell})\}$$

But, a student can only have one advisor. We can express this constraint by requiring that the first argument be a unique key for the advisor relation. We denote this constraint by the notation

$$\texttt{unique\_key}(\texttt{advisor}, [1])!$$

Thus, the first argument of `unique_key` specifies the predicate restricted by the key, and the second argument gives the list of the argument positions that compose the key. An empty list denotes that the derived relation can only contain a single tuple. The exclamation mark is used as the punctuation mark for key constraints. We can now write the following program for our microcollege:

Example 1. For each student select one advisor from professors in the same area

$$\texttt{unique\_key}(\texttt{advisor}, [1])!$$

$$\texttt{advisor}(\texttt{S}, \texttt{P}) \leftarrow \texttt{student}(\texttt{S}, \texttt{Majr}),\ \texttt{professor}(\texttt{P}, \texttt{Majr}).$$

$$\texttt{student}(\texttt{'JimBlack'}, \texttt{ee}).$$
$$\texttt{professor}(\texttt{ohm}, \texttt{ee}).$$
$$\texttt{professor}(\texttt{bell}, \texttt{ee}).$$

Since the key condition ensures that there is only one professor in the resulting advisor table, our query has two possible answers. One is the set

$$\{\texttt{advisor}(\texttt{'JimBlack'}, \texttt{ohm})\}$$

and the other is the set:

$$\{\texttt{advisor}(\texttt{'JimBlack'}, \texttt{bell})\}$$

In the next section, we show that positive programs with keys can be characterized naturally by fixpoint semantics containing multiple canonical answers; in Section 4, we show that their meaning can also be modelled by programs with negated goals under stable models semantics.

Let us consider now some examples that provide a first illustration of the expressive power brought to logic programming by keys in derived relations. The following program constructs a spanning tree rooted in node `a`, for a graph stored in a binary relation `g` as follows:

Example 2. Computing spanning trees

```
unique_key(tree, [2])!
tree(root, a).
tree(Y, Z) ← tree(X, Y), g(Y, Z).

g(a, b).
g(b, c).
g(a, c).
```

Two different spanning trees can be derived, as follows:

$$\{\texttt{tree(root, a)}, \texttt{tree(a, b)}, \texttt{tree(b, c)}\}$$

$$\{\texttt{tree(root, a)}, \texttt{tree(a, b)}, \texttt{tree(a, c)}\}$$

More than one key can be declared for each derived relation. For instance, let us add a second key, `unique_key(tree, [1])`, to the previous graph example. Then, the result may no longer be a spanning tree; instead, it is a simple path, where for each source node, there is only one sink node and vice versa:

Example 3. Computing simple paths

```
unique_key(spath, [1])!
unique_key(spath, [2])!
spath(root, X) ←g(X, Y).
spath(Y, Z) ← spath(X, Y), g(Y, Z).
freenode ← g(_, Y), ¬spath(_, Y).
```

The last rule in Example 3, above, detects whether any node remains free, i.e., whether there is a node not touched by the simple path. Now, a query on whether, for some simple path, there is no free node (i.e., is ¬`freenode` true?) can be used to decide the Hamiltonian path problem for our graph; this is an $\mathcal{NP}$-complete problem. An equivalent way to pose the same question is asking whether `freenode` is true for *all solutions*. A system that generates all possible paths and returns a positive answer when *freenode* holds for all paths implements an *all-answer* semantics. This example illustrates how exponential problems can be expressed in Datalog with keys under this semantics [14].

Polynomial time problems, however, are best treated using *single-answer* semantics, since this can be supported in polynomial time for Datalog programs with key constraints and stratified negation, as discussed later in this paper; moreover, these programs can express all the queries that are polynomial in the size of the database—i.e., the queries in the class *DB-PTIME* [1]. Under single-answer semantics, a deductive system is only expected to compute one out of the many existing canonical models for a program, and return an answer based on this particular model. For certain programs, this approach results in different query answers being returned for different canonical models computed by

the system—nondeterministic queries. For other programs, however, the query answer remains the same for all canonical models—deterministic queries. This is, for instance, the case of the parity query below, which determines whether a non-empty database relation b(X) has an even number of tuples:

Example 4. Counting mod 2

```
unique_key(chain, [1])!
unique_key(chain, [2])!
chain(nil, X) ← b(X).
chain(X, Y) ←    chain(_, X), b(Y).
ca(Y, odd)       chain(nil, Y)
ca(Y, even) ←    ca(X, odd), chain(X, Y).
ca(Y, odd) ←     ca(X, even), chain(X, Y).
mod2(Parity) ← ca(Y, Parity), ¬chain(Y, _).
```

Observe that this program consists of three parts. The first part is the chain rules that enumerate the elements of b(X) one-by-one. The second part is the ca rules that perform a specific aggregate-like computation on the elements of chain—i.e., the odd/even computation for the parity query. The third part is the mod2 rule that uses negation to detect the element of the chain without a successor, and to return the aggregate value 'odd' or 'even' from that of its final element. We will later generalize this pattern to express the computation of generic aggregates.

Observe that the query in Example 4 is deterministic, inasmuch as the answer to the parity question ?mod2(even) is independent of the particular chain being constructed, and only depends on the length of this chain, which is determined by the cardinality of b(x). The parity query is a well-known polynomial query that cannot be answered by Datalog with stratified negation under the genericity assumption [1]. Furthermore, the chain predicate illustrates how the elements of a domain can be arranged in a total order; we thus conclude that negation-stratified Datalog with key constraints can express all *DB-PTIME* queries [1].

In a nutshell, key constraints under single answer semantics extend the expressive power of logic programs, and find important new applications. Of particular importance is the definition of set-aggregates. While aggregates have been used extensively in database applications, particularly in decision support and data mining applications, a general treatment of this fundamental concept had, so far, been lacking and is presented in this paper.

2.1 Basic Definitions

We assume that the reader is familiar with the relational data model and Datalog [1,36].

A logic program P/K consists of a set of rules, P, and a set of key constraints K; each such a constraint has the form unique_key(q, γ), where q is the name of the predicate in P and γ is a subset of the arguments of q. Let I be an

interpretation of P; we say that I satisfies the constraint $\texttt{unique\_key}(\texttt{q}, \gamma)$, when no two atoms in I are identical in all their γ arguments. The notation $I \models K$ will be used to denote that I satisfies every key constraint in K.

The basic semantics of a positive Datalog program P consists of evaluating "in parallel" all applicable instantiations of P's rules. This semantics is formalized by the *Immediate Consequences Operator*, T_P, that defines a mapping over the (Herbrand) interpretations of P, as follows:

$$T_P(I) = \{\ A \mid A \leftarrow B_1, \ldots, B_n\ \in ground(P)\ \wedge\ B_1 \in I \wedge \ldots \wedge B_n \in I\ \}.$$

A rule $r \in ground(P)$ is said to be *enabled* by the interpretation I when all its goals are contained in I. Thus the operator $T_P(I)$ returns the set of the heads of rules enabled by I.

The upward powers of T_P starting from an interpretation I are defined as follows:

$$\begin{aligned} T_P^{\uparrow 0}(I) &= I \\ T_P^{\uparrow (i+1)}(I) &= T_P(T_P^{\uparrow i}(I)), \quad \text{for } i \geq 0 \\ T_P^{\uparrow \omega}(I) &= \bigsqcup_{i \geq 0} T_P^{\uparrow i}(I). \end{aligned}$$

The semantics of a positive program is defined by the least fixpoint of T_P, denoted $lfp(T_P)$, which is also equal to the least model of P, denoted M_P [29]. The least fixpoint of T_p can be computed as the ω-power of T_P applied to the empty set: i.e., $lfp(T_p) = T_P^{\uparrow \omega}(\emptyset)$.

The *inflationary* version of the T_P operator is denoted $\mathbf{T}_P$ and defined as follows:

$$\mathbf{T}_P(I) = T_P(I) \cup I$$

For positive programs, we have:

$$T_P^{\uparrow \omega} = \mathbf{T}_P^{\uparrow \omega} = M_P = lfp(T_P) = lfp(\mathbf{T}_P)$$

The equivalence of model-theoretic and fixpoint semantics no longer holds in Datalog¬ programs, which allow the use of negated goals in rules. Various semantics have therefore been proposed for Datalog¬ programs. For instance, the *inflationary semantics*, which adopts $\mathbf{T}_P^{\uparrow \omega}$ as the meaning of a program P, can be implemented efficiently but lacks desirable logical properties [1]. On the other hand, stratified negation is widely used and combines desirable computational and logical properties [22]; however, stratified negation severely restricts the class of programs that one can write. Formal semantics for more general classes of programs are also available [10,30,2]. Because of its generality and support for nondeterminism, we will use here the *stable model* semantics, that is defined via a *stability transformation* [10], as discussed next. Given an interpretation I and a Datalog¬ program P, the stability transformation derives the positive program $ground_I(P)$ by modifying the rules of $ground(P)$ as follows:

– drop all clauses with a negative literal $\neg A$ in the body with $A \in I$, and
– drop all negative literals in the body of the remaining clauses.

Next, an interpretation M is a stable model for a Datalog¬ program P iff M is the least model of the program $ground_M(P)$. In general, Datalog¬ programs may have zero, one, or many stable models. We shall see how the multiplicity of stable models can be exploited to give a declarative account of non-determinism.

3 Fixpoint Semantics

We use the notation P/K to denote a logic program P constrained by the set of unique keys K.

We make no distinction between interpretations of P and interpretations of P/K; thus every $I \subseteq B_P$ is an interpretation for P/K.

Since a program with key constraints can have multiple interpretations, we will now introduce the concept of family of interpretations. A *family of interpretations* for P is defined as a non-empty set of maximal interpretations for P. More formally:

Definition 1. *Let $\Im$ be a nonempty set of interpretations for P where no element in $\Im$ is a subset of another. Then $\Im$ is called a* family of interpretations *for P.*

The set of families of interpretations for P will be denoted by $fins(P)$.

For instance, let P be the program:

```
a.
b ← a.
```

Then $fins(P)$ consists of the following families of interpretations:

1. $\{\{\}\}$
2. $\{\{a\}\}$
3. $\{\{b\}\}$
3. $\{\{a\},\{b\}\}$
4. $\{\{a,b\}\}$

3.1 Lattice

The $fins(P)$ can be partially ordered as follows:

Definition 2. *Let $\Im_1$ and $\Im_2$ be two elements of $fins(P)$. If $\forall I_1 \in \Im_1, \exists I_2 \in \Im_2$ s.t. $I_1 \subseteq I_2$, then we say that $\Im_1$ is a subfamily of $\Im_2$ and write $\Im_1 \sqsubseteq \Im_2$.*

Now, $(\sqsubseteq, fins(P))$ is a partial order, and also a *complete lattice*, with least upper bound (lub):

$$\Im_1 \sqcup \Im_2 = \{I \in \Im_1 | \neg\exists I_2 \in \Im_2 \text{ s.t. } I_2 \supset I\} \cup \{I \in \Im_2 | \neg\exists I_1 \in \Im_1 \text{ s.t. } I_1 \supseteq I\}$$

The greatest lower bound (glb) is:

$$\Im_1 \sqcap \Im_2 = \{I_1 \cap I_2 | I_1 \in \Im_1, I_2 \in \Im_2 \; and \; \neg(\exists I' \in \Im_1, \exists I'' \in \Im_2 \; s.t. \; I' \cap I'' \supset I_1 \cap I_2)\}$$

These two operations are easily extended to families with infinitely many elements; thus we have a complete lattice, with $\{B_P\}$ as *top* and $\{\emptyset\}$ as *bottom.*

3.2 Fixpoint Semantics of Positive Programs with Keys

Let us consider first the case of positive programs P *without* key constraints, by revisiting the computation of the successive power of T_P, where T_P denotes the immediate consequence operator for P. We will also use the *inflationary* version of this operator, which was previously defined as $\mathbf{T}_P(I) = T_P(I) \cup I$.

The computation $T_P^{\uparrow\omega}(\emptyset) = \mathbf{T}_P^{\uparrow\omega}(\emptyset)$ generates an ascending chain; if I is the result obtained at the last step, the application of $\mathbf{T}_P(I)$ adds to the old I the set of new tuples $T_P(I) - I$, *all at once*. We next define an operator where the new consequences are added one by one; this will be called the *Atomic Consequence Operator (ACO)*, $\mathcal{T}_P$, which is a mapping on families of interpretations. For a singleton set $\{I\}$, $\mathcal{T}_P$ is defined as follows:

$$\mathcal{T}_P(\{I\}) = \{I' \mid \exists x \in [\mathbf{T}_P(I) - I] \; s.t. \; I' = I \cup \{x\}\} \sqcup \{I\}$$

Then, for a family of sets, $\Im$, we have

$$\mathcal{T}_P(\Im) = \bigsqcup_{I \in \Im} \mathcal{T}_P(\{I\})$$

Therefore, our new operator adds to I a single new consequence atom from $\mathbf{T}_P(I) - I$, when this is not empty; thus, it produces a family of interpretations from a singleton interpretation $\{I\}$. When $\mathbf{T}_P(I) = I$, then, by the above definition, $\mathcal{T}_P(\{I\}) = \{I\}$. The following result follows immediately from the definitions:

Proposition 1. *Let P be a positive logic program without keys. Then, $\mathcal{T}_P$ defines a mapping that is monotonic and also continuous.*

Since we have a continuous mapping in a complete lattice, the well-known Knaster-Tarski theorem, and related fixpoint results, can be used to conclude that there always exists solutions of the fixpoint equation $\Im = \mathcal{T}_P(\Im)$, and there also exists the least of such solutions, called the *least fixpoint* of $\mathcal{T}_P$. The least fixpoint of $\mathcal{T}_P$, denoted $lfp(\mathcal{T}_P)$, can be computed as the ω-power of $\mathcal{T}_P$ starting from the bottom element $\{\emptyset\}$.

Proposition 2. *Let P be a positive logic program without key constrains. Then, $\Im = \mathcal{T}_P(\Im)$ has a least fixpoint solution denoted $lfp(\mathcal{T}_P)$, where:*

$$lfp(\mathcal{T}_P) = \mathcal{T}_P^{\uparrow\omega}(\{\emptyset\}) = \bigsqcup_{0<j} \mathcal{T}_P^{\uparrow j}(\{\emptyset\}) = \{lfp(T_P)\}$$

Thus for a positive program without keys, the least fixpoint of the $\mathcal{T}_P$ provides an equivalent characterization of the semantics of positive logic programs since the least fixpoint of $\mathcal{T}_P$ is the singleton set containing the least fixpoint of T_P.

We now consider the situation of a positive program with keys P/K. The Immediate Consequence Operator (ICO) for this program is obtained by simply ignoring the keys: $\mathbf{T}_{P/K}(I) = \mathbf{T}_P(I)$. The ACO is defined as follows:

Definition 3. *Let $T_{P/K}$ be a logic program with key constraints, and let $\{I\} \in fins(P)$ and $\Im \in fins(P)$. Then, $\mathcal{T}_{P/K}(\{I\})$ and $\mathcal{T}_{P/K}(\Im)$ are defined as follows:*

$$\mathcal{T}_{P/K}(\{I\}) = \{I' \mid \exists x \in [\mathbf{T}_P(I) - I] \ s.t. \ \ I' = I \cup \{x\} \ and \ I' \models K\} \sqcup \{I\}$$

$$\mathcal{T}_{P/K}(\Im) = \bigsqcup_{I \in \Im} \mathcal{T}_P(\{I\})$$

For instance, if $\mathcal{T}$ denotes the ACO for our tiny college example, then $\mathcal{T}^{\uparrow 1}(\{\emptyset\})$ is simply a family with three singleton sets, one for each fact in the program:

$$\mathcal{T}^{\uparrow 1}(\{\emptyset\}) = \{\ \{professor(ohm, ee)\}, \{professor(bell, ee)\}, \{student('JimBlack', ee)\}\ \}$$

Thus, $\mathcal{T}^{\uparrow 2}(\{\emptyset\})$ consists of pairs taken from the three program facts:

$$\begin{aligned}\mathcal{T}^{\uparrow 2}(\{\emptyset\}) = \{\ &\{professor(bell, ee), professor(ohm, ee)\}\\ &\{student('JimBlack', ee), professor(bell, ee)\},\\ &\{student('JimBlack', ee), professor(ohm, ee)\}\}\end{aligned}$$

From the first pair, above, we can only obtain a family containing the three original facts; but from the second pair and third pair we obtain two different advisors. In fact, we obtain:

$$\begin{aligned}\mathcal{T}^{\uparrow 3}(\{\emptyset\}) = \{\ &\{student('JimBlack', ee), professor(bell, ee), professor(ohm, ee)\},\\ &\{student('JimBlack', ee), professor(bell, ee),\\ &advisor('JimBlack', bell)\},\\ &\{student('JimBlack', ee), professor(ohm, ee),\\ &advisor('JimBlack', ohm)\}\ \}\end{aligned}$$

In the next step, these three parallel derivations converge into the following two sets:

$$\begin{aligned}\mathcal{T}^{\uparrow 4}(\{\emptyset\}) = \{\ &\{\ student('JimBlack', ee), professor(bell, ee), professor(ohm, ee),\\ &\ \ advisor('JimBlack', bell)\}\\ &\{\ student('JimBlack', ee), professor(bell, ee), professor(ohm, ee),\\ &\ \ advisor('JimBlack', ohm)\}\}\end{aligned}$$

No set can be further enlarged at the next step, given that the addition of a new advisor would violate the key constraints. So we have $\mathcal{T}^{\uparrow 5}(\{\emptyset\}) = \mathcal{T}^{\uparrow 4}(\{\emptyset\})$, and we have reached the fixpoint.

As illustrated by this example, although the operator $\mathcal{T}_{P/K}$ is not monotonic, the ω-power of $\mathcal{T}_{P/K}$ has desirable characteristics that makes it the natural choice for *canonical semantics* of positive programs with keys. In fact we have the following property:

Proposition 3. *Let P/K be a positive program with key constraints. Then, $\mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$ is a fixpoint for $\mathcal{T}_{P/K}$, and each $\{I\} \in \mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$ is a minimal fixpoint for $\mathcal{T}_{P/K}$.*

Proof: The application of $\mathcal{T}_{P/K}$ to $\mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$ can only generate elements which were generated in the ω-derivation. Thus $\mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$ is a fixpoint. Now, let $\{I\} \in \mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$. Clearly, $\mathcal{T}_{P/K}(\{I\}) = \{I\}$, otherwise the previous property does not hold. Thus $\{I\}$ is a fixpoint. To prove that it is minimal, let $J \subset I$. If we trace the derivation chain for $\{I\}$, we find a predecessor of $\{I'\}$ where I' is not a subset of J, but its immediate predecessor, I'' is. Now let $\{x\} = I' - I''$, then $J \cup \{x\}$ does not violate the key constraints (since its superset I does not), and $\{x\}$ is in $\mathbf{T}_P(J)$. Thus $\{J\}$ cannot be a fixpoint. □

Therefore, under the all-answer semantics, we expect the whole family $\mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$ to be returned as the canonical answer, whereas under a single-answer semantics any of the interpretations in $\mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$ is accepted as a valid answer.

In the next section, we introduce an equivalent semantics for our programs with keys using the notion of stable models.

4 Stable-Model Semantics

Programs with keys have an equivalent model-theoretic semantics. We will next show that $\mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$ corresponds to the family of stable models for the program $foe(P/K)$ obtained from P/K by expressing the key constraints by negated goals. The stable model semantics also extends naturally to stratified programs with key constraints.

4.1 Positive Programs with Key Constraints

An equivalent characterization of a positive programs P/K can be obtained by introducing negated goals in the rules of P to enforce the key constraints. The program obtained by this transformation will be denoted $foe(P/K)$, and called the *first order equivalent of* P/K. The program $foe(P/K)$ so obtained always has a formal meaning under stable model semantics [10].

Take, for instance, our advisor example; the rule in Example 1 can also be expressed as follows:

Example 5. The Advisor Example 1 Expressed Using Negation

$$\begin{aligned}
&\texttt{advisor}(\texttt{S},\texttt{P}) \leftarrow && \texttt{student}(\texttt{S},\texttt{Majr},\texttt{Year}),\ \texttt{professor}(\texttt{P},\texttt{Majr}),\\
&&& \neg\texttt{kviol_advisor}(\texttt{S},\texttt{P}).\\
&\texttt{kviol_advisor}(\texttt{S},\texttt{P}) \leftarrow && \texttt{advisor}(\texttt{S},\texttt{P}'),\texttt{P} \neq \texttt{P}'.
\end{aligned}$$

Therefore, we allow a professor P to become the advisor of a student S provided that no other $\texttt{P}' \neq \texttt{P}$ is already an advisor of S. In general, if q is the name of a predicate subject to a key constraint, we use a new predicate kviol_q to denote the violation of key constraints on q; then, we add a kviol_q rule for each key declared for q. Finally, a negated kviol_q goal is added to the original rules defining q. For instance, the simple path program of Example 3 can be re-expressed in the following way:

Example 6. The simple-path program of Example 3 Expressed Using Negation

$$\begin{aligned}
&\texttt{spath}(\texttt{root},\texttt{X}) \leftarrow && \texttt{g}(\texttt{X},\texttt{Y}),\neg\texttt{kviol_spath}(\texttt{root},\texttt{X}).\\
&\texttt{spath}(\texttt{Y},\texttt{Z}) \leftarrow && \texttt{spath}(\texttt{X},\texttt{Y}),\texttt{g}(\texttt{Y},\texttt{Z}),\neg\texttt{kviol_spath}(\texttt{Y},\texttt{Z}).\\
&\texttt{kviol_spath}(\texttt{X1},\texttt{X2}) \leftarrow && \texttt{spath}(\texttt{X1},\texttt{Y2}),\texttt{X2} \neq \texttt{Y2}.\\
&\texttt{kviol_spath}(\texttt{X1},\texttt{X2}) \leftarrow && \texttt{spath}(\texttt{Y1},\texttt{X2}),\texttt{X1} \neq \texttt{Y1}.
\end{aligned}$$

Derivation of $foe(P/K)$. In general, given a program P/K constrained with keys, its first order equivalent $foe(P/K)$ is computed as follows:

1. For each rule r, with head $\texttt{q}(\texttt{Z}_1,\ldots,\texttt{Z}_\texttt{n})$, where q is constrained by some key, add the goal $\neg\texttt{kviol\_q}(\texttt{Z}_1,\ldots,\texttt{Z}_\texttt{n})$ to r,
2. For each $\texttt{unique\_key}(\texttt{q},\texttt{ArgList})!$ in K, where n is the arity of q, add a new rule,

$$\texttt{kviol\_q}(\texttt{X}_1,\ldots,\texttt{X}_\texttt{n}) \leftarrow \texttt{q}(\texttt{Y}_1,\ldots,\texttt{Y}_\texttt{n}),\texttt{Y}_1\theta_1\texttt{X}_1,\ \ldots,\ \texttt{Y}_\texttt{n}\theta_\texttt{n}\texttt{X}_\texttt{n}.$$

where θ_j denotes the equality symbol '=' for every j in ArgList, and the inequality symbol '$\neq$' for every j not in ArgList.

For instance, the foe of our advisor example is:

$$\begin{aligned}
&\texttt{advisor}(\texttt{S},\texttt{P}) \leftarrow && \texttt{student}(\texttt{S},\texttt{Majr},\texttt{Year}),\ \texttt{professor}(\texttt{P},\texttt{Majr}),\\
&&& \neg\texttt{kviol_advisor}(\texttt{S},\texttt{P}).\\
&\texttt{kviol_advisor}(\texttt{X}_1,\texttt{X}_2) \leftarrow && \texttt{advisor}(\texttt{Y}_1,\texttt{Y}_2),\texttt{X}_1 = \texttt{Y}_1,\texttt{X}_2 \neq \texttt{Y}_2.
\end{aligned}$$

This transformation does in fact produce the rules of Example 6, after we replace equals with equals and eliminate all equality goals. The newly introduced predicates with the prefix kviol will be called *key-violation predicates.*

Stable models provide the formal semantics for our foe programs:

Proposition 4. *Let P/K be a positive logic program with keys. Then $foe(P/K)$ has one or more stable models.*

A proof for this proposition can be easily derived from [25,13], where the same transformation is used to define the formal semantics of programs with the `choice` construct.

With I an interpretation of $foe(P)$, let $pos(I)$ denote the interpretation obtained by removing all the key-violation atoms from I and leaving the others unchanged. Likewise, if $\Im$ is a family of interpretation of $foe(P)$, then we define:

$$pos(\Im) = \bigsqcup_{I \in \Im} pos(I)$$

Then, the following theorem elucidates the equivalence between the two semantics:

Proposition 5. *Let P/K be a positive program, and Σ be the set of stable models for $foe(P/K)$. Then $pos(\Sigma) = \mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$.*

Proof: Let $I \in \mathcal{T}^{\uparrow\omega}_{P}(\{\emptyset\})$, and $P_I = ground_I(foe(P/K))$ be the program produced by the stability transformation on $foe(P/K)$. It suffices to show that $T^{\uparrow\omega}_{P_I}(\{\emptyset\}) = I$, i.e., that $\{I\} = \mathcal{T}^{\uparrow\omega}_{P_I}(\{\emptyset\})$. Now, take a derivation in $\mathcal{T}^{\uparrow\omega}_{P/K}(\{\emptyset\})$ producing I; we can find an identical derivation in $\mathcal{T}^{\uparrow\omega}_{P_I}(\{\emptyset\})$. This concludes our proof. □

4.2 Stratification

The notion of stratification significantly increases the expressive power of Datalog, while retaining the declarative fixpoint semantics of programs. Consider first the notion of stratification with respect to negation for programs without key constraints:

Definition 4. *Let P be a program with negated goals, and $\sigma_1, \ldots, \sigma_n$ be a partition of the predicate names in P. Then, P is said to be stratified, when for each rule $r \in P$ (with head h_r) and each goal g_r in r, the following property holds:*

1. *$stratum(h_r) > stratum(g_r)$ if g_r is a negated goal*
2. *$stratum(h_r) \geq stratum(g_r)$ if g_r is a positive goal.*

Therefore, a stratified program P can be viewed as a stack of rule layers, where the higher layers do not influence the lower ones. Thus the correct semantics can be assigned to a program by starting from the bottom layer and proceeding upward, with the understanding that computation for the higher layers cannot affect lower ones.

The computation can be implemented using the ICO T_P, which, in the presence of negated goals, is generalized as follows. A rule $r \in ground(P)$ is said to be *enabled* by an interpretation I when all of its positive goals are in I and none of its negated goals are in I. Then, $T_P(I)$ is defined as containing the heads of all rules in $ground(P)$ that are enabled by I. (This change automatically adjusts the definitions of $\mathbf{T}$ and $\mathcal{T}$ that are based on T_P.)

Therefore, let $I[\leq j]$ and $P[\leq j]$, respectively, denote the atoms in I and the rules in P whose head belongs to strata $\leq j$. Also let $P[j]$ denote the set of rules in P whose head belongs to stratum j. Then, we observe that for a stratified program P, the mapping defined by $P[j]$ (i.e., $T_{P[j]}$) is *monotonic with respect* to $I[j]$. Thus, if I_{j-1} is the meaning of $P[\leq j-1]$, then $\mathbf{T}^{\uparrow\omega}_{P[j]}(I_{j-1})$ is the meaning of $P[\leq j]$.

Thus, let P be a program stratified with respect to negation and without key constraints; then the following algorithm inductively constructs the iterated fixpoint for T_P (and $\mathbf{T}_P$):

Iterated Fixpoint computation for $\mathbf{T}_P$, where P is stratified with strata $\sigma_1, \ldots, \sigma_n$.

1. Let $I_0 = \emptyset$;
2. For $j = 1, \ldots, n$, let $I_j = \mathbf{T}^{\uparrow\omega}_{P[j]}(I_{j-1})$

For every $1 \leq j \leq n$, $I_j = I_n[\leq j]$ is a minimal fixpoint of $P[\leq j]$. The interpretation I_n obtained at the end of this computation is called the iterated fixpoint for T_P and defines the meaning of the program P. It is well-known that the iterated fixpoint for a stratified program P is equal to P's unique stable model [36].

These notions can now be naturally extended to programs with key constraints. A program P/K is stratified whenever its keyless counterpart P is stratified. Let $P/K[j]$ denote the rules with head in the j^{th} stratum, along with the key constraints on their head predicates; also, let $P/K[\leq j]$ denote the rules with head in strata lower than the j^{th} stratum, along with their applicable key constraints. Finally, let:

$$\Im[\leq j] = \bigsqcup_{I \in \Im} I[\leq j]$$

The notion of $\mathcal{T}$ can be extended in natural fashion to stratified programs. If $\Im_{j-1}$ is the meaning of $P/K[\leq j-1]$, then $\mathcal{T}^{\uparrow\omega}_{P[j]}(\Im_{j-1})$ is the natural meaning of $P/K[\leq j]$.

Thus we have the following extension of the iterated fixpoint algorithm:

Iterated Fixpoint Computation for $\mathcal{T}_{P/K}$ where P/K is stratified with strata $\sigma_1, \ldots, \sigma_n$.

1. Let $\Im_0 = \{\emptyset\}$;
2. For $j = 1, \ldots, n$, let $\Im_j = \mathcal{T}^{\uparrow\omega}_{P/K[j]}(\Im_{j-1})$

The family of interpretations $\Im_n$ obtained from this computation will be called the *iterated fixpoint* for $\mathcal{T}_{P/K}$. The iterated fixpoint for $\mathcal{T}_{P/K}$ defines the meaning of P/K; it has the property that, for each $1 \leq j \leq n$, each member in $\Im_j = \Im_n[\leq j]$ is a minimal fixpoint for $\mathcal{T}_{P/K[\leq j]}$.

Stable Model Semantics for Stratified Programs. Every program P that is stratified with respect to negation has a unique stable model that can be computed by the iterated fixpoint computation for $\mathbf{T}_P$ previously discussed. Likewise, every stratified program P/K can be expanded into its first order equivalent $foe(P/K)$. Then, it can be shown that (i) $foe(P/K)$ always has one or more stable models, and (ii) if Σ denotes the family of its stable models, then $pos(\Sigma)$ coincides with the iterated fixpoint of $\mathcal{T}_{P/K}$.

5 Single-Answer Semantics and Nondeterminism

The derivation $\mathcal{T}_{P/K}^{\uparrow\omega}(\{\emptyset\})$ can be used to compute in parallel all the stable models for a positive program $foe(P/K)$. In this computation, each application of $\mathcal{T}_{P/K}$ expands in parallel all interpretations in the current family, by the addition of a single new element to each interpretation. In [38], we discuss *condensed derivations* based on $\mathcal{T}_{P/K}$, which accelerate the derivation process by adding several new elements at each step of the computation. This ensures a faster convergence toward the final result, while still computing all stable models at once. Even with condensed derivations, the computation of all stable models requires exponential time, since the number of such models can be exponential in the size of the database. This, computational complexity might be acceptable when dealing with $\mathcal{NP}$-complete problems, such as deciding the existence of an Hamiltonian path. However, in many situations involving programs with multiple stable models, *only one* such model, not all of them, is required in practice. For instance, this is the case of Example 4, where we use choice to enumerate into a chain the elements of a set one by one, with the knowledge that the even/odd parity of the whole set only depends on its cardinality, and not on the particular chain used. Therefore for Example 4, the computation of any stable model will suffice to answer correctly the parity query. Since this situation is common for many queries, we need efficient operators for computing a single stable model.

Even with $\mathcal{NP}$-complete problems, it is normally desirable to generate the stable models in a serial rather than parallel fashion. For instance, for the Hamiltonian circuit problem of Example 3, we can test if the last generated model satisfies the desired property (i.e., if there is any `freenode`), and only if this test fails, proceed with the generation of another model— normally, calling on some heuristics to aid in the search for a good model. On the average, this search succeeds without having to produce an exponential number of stable models, since exponential complexity only represents the worst-case behavior for many $\mathcal{NP}$-complete algorithms.

Now, the computation of a single stable model is in general $\mathcal{NP}$-hard [26]; however, this computation for a program $foe(P/K)$ derived from one with key constraints can be performed in polynomial time, and, as we describe next, with minimal overhead with respect to the standard fixpoint computation. Therefore, we next concentrate on the problem of generating a single element in $\mathcal{T}_{P/K}^{\uparrow\omega}(\{\emptyset\})$, and on expressing polynomial-time queries using this single-answer semantics.

We define next the notions of *soundness* and *completeness* for nondeterministic operators to be used to compute an element in $\mathcal{T}_P^{\uparrow\omega}(\{\emptyset\})$.

Definition 5. *Let P/K be a logic program with keys, and $\mathcal{C}$ be a class of functions on interpretations of P. Then we define the following two properties:*

1. Soundness. *A function $\tau \in \mathcal{C}$ will be said to be sound for a program P/K when $\tau^{\uparrow\omega}(\emptyset) \in \mathcal{T}_{P/K}^{\uparrow\omega}(\{\emptyset\})$. The function class $\mathcal{C}$ will be said to be sound when all its members are sound.*
2. Completeness. *The function class $\mathcal{C}$ will be said to be complete for a program $\mathcal{T}_{P/K}$ when for each $M \in \mathcal{T}_{P/K}^{\uparrow\omega}(\{\emptyset\})$ there exists some $\tau \in \mathcal{C}$ such that: $\tau^{\uparrow\omega}(\emptyset) = M$.*

In situations where any answer will solve the problem at hand, there is no point in seeking completeness and we can limit ourselves to classes of functions that are sound, and efficient to compute, even if completeness is lost; *eager derivations* discussed next represent an interesting class of such functions.

Definition 6. *Let P/K be a program with key constraints, and let $\Gamma(I)$ be a function on interpretations of P. Then, $\Gamma(I)$ will be called an* eager derivation *operator for P/K if it satisfies the following three conditions:*

1. $I \subseteq \Gamma(I) \subseteq \mathbf{T}_P(I)$
2. $\Gamma(I) \models K$
3. *Every subset of $\mathbf{T}_P(I)$ that is a proper superset of $\Gamma(I)$ violates some key constraint in K.*

Let $\mathcal{C}_\Gamma$ be the class of eager derivation operators for a given program P/K. Then it is immediate to see that $\mathcal{C}_\Gamma$ is sound for all programs.

Eager derivation operators can be implemented easily. Their implementation only requires tables to memorize atoms previously derived and compare the new values against previous ones to avoid key violations. Inasmuch as table-based memorization is already part of the basic mechanism for the computation of fixpoints in deductive databases, key constraints are easy to implement.

A limitation of eager derivation operators is that they do not form a complete class for all positive programs with key constraints. This topic is discussed in [38], where classes of operators which are both sound and complete are also discussed. However, in the rest of this paper, we only use key constraints to define chain rules, such as those in Example 4; for these rules, the eager derivations are *complete*—in addition to being sound and efficiently computable.

6 Set Aggregates in Logic

The additional expressive power brought to Datalog by key constraints finds many uses; here we employ it to achieve a formal characterization of database aggregates, thus solving an important open problem in database theory and logic

programming. In fact, the state-of-the-art characterization of aggregates relies on the assumption that the universe is totally ordered [36]. Using this assumption, the atoms satisfying a given predicate are chained together in ascending order, starting from the least value and ending with the largest value. Unfortunately, this solution has four serious drawbacks, since (i) it compromises data independence by violating the genericity property [1], (ii) it relies on negation, thus infecting aggregates with the nonmonotonic curse, (iii) it is often inefficient since it requires the data to be sorted before aggregation, and (iv) it cannot be applied to more advanced forms of aggregation, such as on-line aggregates and rollups, that are used in decision support and other advanced applications [33].

Online aggregation [8], in particular, cannot be expressed under the current approach that relies on a totally ordered universe to sort the elements of the set being processed, starting from its least element. In fact, at the core of on-line aggregation, there is the idea of returning partial results after visiting a proper subset of the given dataset, while the rest is still unknown. Now, it is impossible to compute the least element of a set when only part of it is known.

We next show that all these problems find a simple solution once key constraints are added to Datalog. For concreteness, we use the aggregate constructs of $\mathcal{LDL}$++ [4], but very similar syntactic constructs are used by other systems (e.g., *CORAL* [23]), and the semantics here proposed is general and applicable to every logic-based language and database query language.

6.1 User Defined Aggregates

Consider the parity query of Example 4. To define an equivalent parity aggregate in $\mathcal{LDL}$++ the user will write the following rules:

Example 7. Definition rules for the parity aggregate `mod2`

```
single(mod2, _, odd).
multi(mod2, X, odd, even).
multi(mod2, X, even, odd).
freturn(mod2, _, Parity, Parity).
```

These rules have the same function as the last four rules in Example 4. The `single` rule specifies how to initialize the computation of the `mod2` aggregate by specifying its value on a singleton set (same as the first `ca` rule in the example). The two `multi` rules instead specify how the new aggregate value (the fourth argument) should be updated for each new input value (second argument), given its previous value (third argument). (Thus these rules perform the same function as the second and the third of the `ca` rules in Example 4.) The `freturn` rule specifies (as fourth argument) the value to be returned once the last element in the set is detected (same as the last rule in Example 4). For `mod2`, the value returned is simply taken from the third argument, where it was left by the multi rule executed on the last element of the set. Two important observations can therefore be made:

1. We have described a very general method for defining aggregates by specifying the computation to be performed upon (i) the initial value, (ii) each successive value, and (iii) the final value in the set. This paradigm is very general, and also describes the mechanism for introducing user defined aggregates (UDAs) used by SQL3 and in the AXL system [33].
2. The correspondence between the above rules and those of Example 4 outlines the possibility of providing a logic semantics to UDAs by simply expanding the `single`, `multi`, and `freturn` rules into an equivalent logic program (using the chain rules) such as that of Example 4.

The rules in Example 7 are generic, and can be applied to any set of facts. To reproduce the behavior of Example 4, they must be applied to `b(X)`. In $\mathcal{LDL}$++ this is specified by the aggregate-invocation rule:

$$\texttt{p}(\texttt{mod2}\langle\texttt{X}\rangle) \leftarrow \texttt{b}(\texttt{X}).$$

that specifies that the result of the computation of `mod2` on `b(X)` is returned as the argument of a predicate, that our user has named `p`.

There has been much recent interest in online aggregates [8], which also find important applications in logic programming, as discussed later in this paper. For instance, when computing averages on non-skewed data, the aggregate often converges toward the final value long before all the elements in the set are visited. Thus, the system should support *early returns* to allow the user to check convergence and stop the computation as soon as the series of successive values has converged within the prescribed accuracy [8]. UDAs with early returns can be defined in $\mathcal{LDL}$++ through the use of `ereturn` rules.

Say, for instance, that we want to define a new aggregate `myavg`, and apply it to the elements of `d(Y)`, and view the results of this computation as a predicate `q`. Then, the $\mathcal{LDL}$++ programmer must specify one *aggregate-application rule*, and several *aggregate-definition rules*. For instance, the following is an aggregate application rule:

$$\texttt{r}:\ \texttt{q}(\texttt{myavg}\langle\texttt{Y}\rangle) \leftarrow \texttt{d}(\texttt{Y}).$$

The $\langle \ldots \rangle$ notation in the head of r denotes an aggregate; this rule specifies that the definition rules for `myavg` must be applied to the stream of `Y`-values that satisfy the body of the rule.

The aggregate definition rules include: (i) *single* rule(s) (ii) *multi* rule(s), (iii) *freturn* rule(s) for final returns and/or (iv) *ereturn* rule(s) for early returns. All four kinds of rules are used in the following definition of `myavgr`:

$$\texttt{single}(\texttt{myavg}, \texttt{Y}, \texttt{cs}(\texttt{1}, \texttt{Y})).$$
$$\texttt{multi}(\texttt{myavg}, \texttt{Y}, \texttt{cs}(\texttt{Cnt}, \texttt{Sum}), \texttt{cs}(\texttt{Cnt1}, \texttt{Sum1})) \leftarrow \texttt{Cnt1} = \texttt{Cnt} + \texttt{1}, \texttt{Sum1} = \texttt{Sum} + \texttt{Y}.$$

$$\texttt{freturn}(\texttt{myavg}, \texttt{Y}, \texttt{cs}(\texttt{Cnt}, \texttt{Sum}), \texttt{Val}) \leftarrow \texttt{Val} = \texttt{Sum}/\texttt{Cnt}.$$

$$\texttt{ereturn}(\texttt{myavg}, \texttt{X}, (\texttt{Sum}, \texttt{Count}), \texttt{Avg}) \leftarrow \\ \texttt{Count mod 100} = 0, \texttt{Avg} = \texttt{Sum}/\texttt{Count}.$$

Observe that the first argument in the head of the single, multi, ereturn, and freturn rules contains the name of the aggregate: therefore, these aggregate definition rules can only be used by aggregate application rules that contain $\texttt{myavg}\langle \ldots \rangle$ in the head.

The second argument in the head of a single or multi rule holds the 'new' value from the input stream, while the last argument holds the partial value returned by the previous computation. Thus, for averages, the last argument should hold the pair $\texttt{cs}(\texttt{Count}, \texttt{Sum})$. The single rule specifies the value of the aggregate for a singleton set (containing the first value in the stream); for myavg, the singleton rule must return $\texttt{cs}(1, \texttt{Y})$. The multi rules prescribe an inductive computation on a set with $n+1$ elements, by specifying how the $n+1^{th}$ element in the stream is to be combined with the value returned (as third argument in multi) by the computation on the first n elements. For myavg, the count is increased by one and the sum is increased by the new value in the stream.

The freturn rules specify how the final value(s) of the aggregate are to be returned. For myavg, we return the ratio of sum and count. The ereturn rules specify when early returns are to be produced and what are their values. In particular for myavg, we produce early returns every 100 elements in the stream, and the value produced is the current ratio sum/count—online aggregation.

6.2 Semantics of Aggregates

In general, the semantics of an aggregate application rule r

$$\texttt{r}: \ \texttt{q}(\texttt{myavg}\langle \texttt{Y} \rangle) \leftarrow \texttt{d}(\texttt{Y}).$$

can be defined by expanding it into its *key-constrained equivalent* logic program, denoted $kce(r)$, which contains the following rules:

1. A *main rule*
$$\texttt{p}(\texttt{Y}) \leftarrow \texttt{results}(\texttt{avg}, \texttt{Y}).$$
where $\texttt{results}(\texttt{avg}, \texttt{Y})$ is derived from $\texttt{d}(\texttt{Y})$ by a program consisting of:
2. The *chain rules* that link the elements of $\texttt{d}(\texttt{Y})$ into an order-inducing chain (nil is a special value not in $\texttt{d}(\texttt{Y})$),
$$\begin{aligned} &\texttt{unique\_key}(\texttt{chain}_\texttt{r}, [1])! \\ &\texttt{unique\_key}(\texttt{chain}_\texttt{r}, [2])! \\ &\texttt{chain}_\texttt{r}(\texttt{nil}, \texttt{Y}) \leftarrow \ \texttt{d}(\texttt{Y}). \\ &\texttt{chain}_\texttt{r}(\texttt{Y}, \texttt{Z}) \leftarrow \quad \texttt{chain}_\texttt{r}(\texttt{X}, \texttt{Y}), \texttt{d}(\texttt{Z}). \end{aligned}$$
3. The cagr rules that perform the inductive computation:
$$\begin{aligned} &\texttt{cagr}(\texttt{AgName}, \texttt{Y}, \texttt{New}) \leftarrow \ \texttt{chain}_\texttt{r}(\texttt{nil}, \texttt{Y}), \texttt{Y} \neq \texttt{nil}, \texttt{single}(\texttt{myagr}, \texttt{Y}, \texttt{New}). \\ &\texttt{cagr}(\texttt{AgName}, \texttt{Y2}, \texttt{New}) \leftarrow \texttt{chain}_\texttt{r}(\texttt{Y1}, \texttt{Y2}), \texttt{cagr}(\texttt{AgName}, \texttt{Y1}, \texttt{Old}), \\ &\qquad\qquad\qquad\qquad\quad \texttt{multi}(\texttt{AgName}, \texttt{Y2}, \texttt{Old}, \texttt{New}). \end{aligned}$$

Thus, the `cagr` rules are used to memorize the previous results, and to apply (i) `single` to the first element of `d(Y)` (i.e., for the pattern $\texttt{chain}_\texttt{r}(\texttt{nil}, \texttt{Y})$) and (ii) `multi` to the successive elements.

4. The two `results` rules, where the first rule produces *early returns* and second rule produces *final returns* as follows:

$$\begin{aligned}\texttt{results}(\texttt{AgName}, \texttt{Y2}, \texttt{New}) \leftarrow\ & \texttt{chain}_\texttt{r}(\texttt{Y1}, \texttt{Y2}), \texttt{cagr}(\texttt{AgName}, \texttt{Y1}, \texttt{Old}),\\ & \texttt{ereturn}(\texttt{AgName}, \texttt{Y2}, \texttt{Old}, \texttt{Yield}).\\ \texttt{results}(\texttt{AgName}, \texttt{AgValue}) \leftarrow\ & \texttt{chain}_\texttt{r}(\texttt{X}, \texttt{Y}), \neg\texttt{chain}_\texttt{r}(\texttt{Y}, \_),\\ & \texttt{cagr}(\texttt{AgName}, \texttt{Y}, \texttt{Old}),\\ & \texttt{freturn}(\texttt{AgName}, \texttt{Y}, \texttt{Old}, \texttt{AgValue}).\end{aligned}$$

Therefore, the first `results` rule produces the early returns by applying `ereturn` to every element in the chain, and the second rule produces the final returns by applying `freturn` on the last element in the chain (i.e., the element without a successor).

In $\mathcal{LDL}$++, an implicit group-by operation is performed on the head arguments not used to apply aggregates. Thus, to compute the average salary of employees grouped by `Dno`, the user can write:

$$\texttt{avgsal}(\texttt{Dno}, \texttt{myavg}\langle\texttt{Sal}\rangle) \leftarrow \texttt{emp}(\texttt{Eno}, \texttt{Sal}, \texttt{Dno}).$$

As discussed in [34], the semantics of aggregates with group-by can simply be defined by including an additional argument in the predicates $\texttt{chain}_\texttt{r}$ and `results` to hold the group-by attributes.

6.3 Applications of User Defined Aggregates

We will now discuss the use of UDAs to express polynomial algorithms in a natural and efficient way. These algorithms use aggregates in programs that yield the correct final results unaffected by the nondeterministic behavior of the aggregates. Therefore, aggregate computation here uses single-answer semantics, which assures polynomial complexity.

Let us consider first uses of nonmonotonic aggregates. For instance, say that from a set of pairs such as (`Name`, `YearOfBirth`) as input, we want to return the `Name` of the youngest person (i.e., the person born in the latest year). This computation cannot be expressed directly as an aggregate in SQL, but can be expressed by the UDA `youngest` given below (in $\mathcal{LDL}$++, a vector of n arguments $(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n})$ is basically treated as a n-argument function with a default name).

$$\begin{aligned}&\texttt{single}(\texttt{youngest},\ (\texttt{N}, \texttt{Y}), (\texttt{N}, \texttt{Y})).\\ &\texttt{multi}\ (\texttt{youngest},\ (\texttt{N}, \texttt{Y}), (\texttt{N1}, \texttt{Y1}), (\texttt{N}, \texttt{Y})) \leftarrow \quad \texttt{Y} \geq \texttt{Y1}.\\ &\texttt{multi}\ (\texttt{youngest},\ (\texttt{N}, \texttt{Y}), (\texttt{N1}, \texttt{Y1}), (\texttt{N1}, \texttt{Y1})) \leftarrow \texttt{Y} \leq \texttt{Y1}.\\ &\texttt{freturn}(\texttt{youngest}, (\texttt{N}, \texttt{Y}), (\texttt{N1}, \texttt{Y1}), \texttt{N1}).\end{aligned}$$

User-defined aggregates provide a simple solution to a number of complex problems in deductive databases; due to space limitations we will here consider only simple examples—a more complete set of examples can be found in [37].

We already discussed the definition and uses of online aggregates, such as myavg that returns values every 100 samples. In a more general framework, the user would want to control how often new results are to be returned to the user, on the basis of the estimated progress toward convergence in the computation [8]. UDAs provide a natural setting for this level of control.

Applications of UDAs are too many to mention. But for an example, take the interval coalescing problem of temporal databases [35]. For instance, say that from a base relation emp(Eno, Sal, Dept, (From, To)), we project out the attribute Sal and Dept; then the same Eno appears in tuples with overlapping valid-time intervals and must be coalesced. Here we use closed intervals represented by the pair (From, To) where From is the start-time, and To is the end-time. Under the assumption that tuples are sorted by increasing start-time, we can use a special coales aggregate to perform the task in one pass through the data.

Example 8. Coalescing overlapping intervals sorted by start time.

$$\begin{aligned}
&\texttt{empProj}(\texttt{Eno}, \texttt{coales}\langle(\texttt{From}, \texttt{To})\rangle) \leftarrow \quad \texttt{emp}(\texttt{Eno}, _, _, (\texttt{From}, \texttt{To})).\\
&\texttt{single}(\texttt{coales}, (\texttt{Frm}, \texttt{To}), (\texttt{Frm}, \texttt{To})).\\
&\texttt{multi}(\texttt{coales}, (\texttt{Nfr}, \texttt{Nto}), (\texttt{Cfr}, \texttt{Cto}), (\texttt{Cfr}, \texttt{Lgr})) \leftarrow \quad \texttt{Nfr} \leq \texttt{Cto},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{larger}(\texttt{Cto}, \texttt{Nto}, \texttt{Lgr}).\\
&\texttt{multi}(\texttt{coales}, (\texttt{Nfr}, \texttt{Nto}), (\texttt{Cfr}, \texttt{Cto}), (\texttt{Cfr}, \texttt{Nto})) \leftarrow \quad \texttt{Nfr} > \texttt{Cto}.\\
&\texttt{ereturn}(\texttt{coales}, (\texttt{Nfr}, \texttt{Nto}), (\texttt{Cfr}, \texttt{Cto}), (\texttt{Cfr}, \texttt{Cto})) \leftarrow \texttt{Nfr} > \texttt{Cto}.\\
&\texttt{freturn}(\texttt{coales}, _, \texttt{LastInt}, \texttt{LastInt}).\\
&\texttt{larger}(\texttt{X}, \texttt{Y}, \texttt{X}) \leftarrow \quad \texttt{X} \geq \texttt{Y}.\\
&\texttt{larger}(\texttt{X}, \texttt{Y}, \texttt{X}) \leftarrow \quad \texttt{X} < \texttt{Y}.
\end{aligned}$$

Thus, the single rule starts the coalescing process by setting the current interval equal to the first interval. The multi rule operates as follows: when the new interval (Nfr, Nto) overlaps the current interval (Cfr, Cto) (i.e., when Nfr ≤ Cto), the two are coalesced into an interval that begins at Cfr, and ends with the larger of Nto and Cto; otherwise, the current interval is returned and the new interval becomes the current one.

7 Monotonicity

Commercial database systems and most deductive database systems disallow the use of aggregates in recursion and require programs to be stratified with respect to aggregates. This restriction is also part of the SQL99 standards [7]. However, many important algorithms, particularly greedy algorithms, use aggregates such as count, sum, min and max in a monotonic fashion, inasmuch as previous results are never discarded. This observation has inspired a significant amount of previous work seeking efficient expression of these algorithms in logic [27,6,24,31,9,15]. At the core of this issue there is the characterization of programs where aggregates behave monotonically and can therefore be freely used in recursion. For many interesting programs, special lattices can be found

in which aggregates are monotonic [24]. But the identification of such lattices cannot be automated [31], nor is the computation of fixpoints for such programs. Our newly introduced theory of aggregates provides a definitive solution to the monotonic aggregation problem, including a simple syntactic characterization to determine if an aggregate is monotonic and can thus be used freely in recursion.

7.1 Partial Monotonicity

For a program P/K, we will use the words *constrained predicates* and *free predicates* to denote predicates that are constrained by keys and those that are not. With I an interpretation, let I_c, and I_f, respectively, denote the atoms in I that are instances of constrained and free predicates; I_c will be called the *constrained component* of I, and I_f is called the *free component* of I. Then, let I and J be two interpretations such that $I \subseteq J$ and $I_c = J_c$ (thus $I_f \subseteq J_f$). Likewise, each family $\Im$ can be partitioned into the family of its constrained components, $\Im_c$, and the family of its free components, $\Im_f$.

Then, the following proposition shows that a program P/K defines a monotonic transformation with respect to the free components of families of interpretations:

Proposition 6. Partial Monotonicity: *Let* $\Im$ *and* $\Im'$ *be two families of interpretations for a program* P/K. *If* $\Im \sqsubseteq \Im'$, *while* $\Im_c = \Im'_c$ *then,* $\mathcal{T}_{P/K}(\Im) \sqsubseteq \mathcal{T}_{P/K}(\Im')$.

Proof. It suffices to prove the property for two singleton sets $\{I\}$ and $\{J\}$ where $I_f \subseteq J_f$, while $I_c = J_c$. Take an arbitrary $I' \in \mathcal{T}_{P/K}(\{I\})$: we need to show that there exists a $J' \in \mathcal{T}_{P/K}(\{J\})$ where $I' \subseteq J'$. If $I' \subseteq J$ the conclusion is trivial; else, let $I' = I \cup \{x\}$, $x \in T_P(I) - I$, and $I' \models K$. Since I is a subset of J but I' is not, x is not in J, and $x \in T_P(J) - J$. Also, if $J' = J \cup \{x\}$, $J' \models K$ (since $J'_c = I'_c$). Thus, $J' \in \mathcal{T}_{P/K}(\{J\})$. □

This partial monotonicity property (i.e., monotonicity w.r.t. free predicates only) extends to the successive powers of $\mathcal{T}_{P/K}$, including its ω-power. Thus If $\Im \sqsubseteq \Im'$, while $\Im_c = \Im'_c$ then, $\mathcal{T}^{\uparrow\omega}_{P/K}(\Im) \sqsubseteq \mathcal{T}^{\uparrow\omega}_{P/K}(\Im')$. This result shows that the program P/K defines a monotonic mapping from unconstrained predicates to every other predicate in the program. It is customary in deductive databases to draw a distinction between extensional information (base relations) and intensional information (derived relations). Therefore, a program can be viewed as defining a mapping from base relations to derived relations. Therefore, the partial monotonicity property states that the mapping from database relations free of key constraints to derived relations is monotonic—i.e., the larger the base relations, the larger the derived relations.

For a base relation R that is constrained by keys, we can introduce an auxiliary input relation R_I free of key constraints, along with a copy rule that derives R from R_I. Then, we can view R_I as the input relation and R as a result of filtering R_I with the key constraints. Then, we have a monotonic mapping from the input relation R_I to the derived relations in the program.

7.2 Monotonic Aggregates

Users normally think of an aggregate application rule, such as r, as a direct mapping from r's body to r's head—a mapping which behaves according to the rules defining the aggregate. This view is also close to the actual implementation, since in a system such as $\mathcal{LDL}$++ the execution of the rules in $kce(r)$ is already built into the system.

The *encapsulate program* for an aggregate application rule r, will be denoted $\epsilon(r)$ and contains all the rules in $kce(r)$ and the single, multi, ereturn and freturn rules defining the aggregates used in r. Then, the *transitive* mapping defined by $\epsilon(r)$ transforms families of interpretations of the body of r to families of interpretations of the heads of rules in $\epsilon(r)$. With I an interpretation of the body of r (i.e., a set of atoms from predicates in the body of r), then the mapping for $\epsilon(r)$ is equal to $\mathcal{T}^{\uparrow\omega}_{\epsilon(r)}(\{I\})$, when there are no `freturn` rules, and is equal to the result of the iterated fixpoint of the stratified $\epsilon(r)$ program, otherwise.

For instance, consider the definition and application rules for an online count aggregate `msum`:

$$\mathtt{r'}: \mathtt{q}(\mathtt{msum}\langle \mathtt{X}\rangle) \leftarrow \mathtt{p}(\mathtt{X}).$$

$$\begin{aligned}
&\mathtt{single}(\mathtt{msum}, \mathtt{Y}, \mathtt{Y}).\\
&\mathtt{multi}(\mathtt{msum}, \mathtt{Y}, \mathtt{Old}, \mathtt{New}) \leftarrow \quad \mathtt{New} = \mathtt{Old} + \mathtt{Y}.\\
&\mathtt{ereturn}(\mathtt{msum}, \mathtt{Y}, \mathtt{Old}, \mathtt{New}) \leftarrow \mathtt{New} = \mathtt{Old} + \mathtt{Y}.
\end{aligned}$$

The transitive mapping established by $\epsilon(r')$ can be summarized by the $\mathtt{chain}_\mathtt{r}$ atoms, which describe a particular sequencing of the elements in I and the aggregate values for the sequence so generated:

$\Im$	$\mathcal{T}^{\uparrow\omega}_{\epsilon(r')}(\Im)$
$\{\{p(3)\}\}$	$\{\{chain_r(nil, 3), q(3)\}\}$
$\{\{p(1), p(3)\}\}$	$\{\{chain_r(nil, 1), chain_r(1, 3), q(1), q(4)\},$ $\{chain_r(nil, 3), chain_r(3, 1), q(3), q(4)\}\}$
...	...

Therefore, the mapping defined by the aggregate rules is multivalued —i.e., from families of interpretations to families of interpretations. The ICO for the set of non aggregate rules P' can also be seen as a mapping between families of interpretations by simply letting $T_{P'}(\{I\}) = \{T_{P'}(I)\}$. Then, the *encapsulated consequence operator* for a program with aggregates combines the immediate consequence operator for regular rules with the transitive consequences for the aggregate rules. Because of the partial monotonicity properties of programs with key constraints, we now derive the following property:

Proposition 7. *Let P be a positive program with aggregates defined without final return rules. Then, the encapsulated consequence operator for P is monotonic in the lattice of families of interpretations.*

Therefore, aggregates defined without `freturn` rules will be called *monotonic*; thus, monotonic aggregates can be used freely in recursive programs. Aggregate computation in actual programs is very similar to the seminaive computation used to implement deductive databases [5,35], which is based on combining old values with new values according to rules obtained by the symbolic differentiation of the original rules. For aggregates, we can use the same framework with the difference that the rules for storing the old values and those for producing the results are now given explicitly by the programmer through the single/multi and ereturn/freturn rules in the definition.

7.3 Aggregates in Recursion

Our newly introduced theory of aggregates provides a definitive solution to the monotonic aggregation problem, with a simple syntactic criterion to decide if an aggregate is monotonic and can thus be used freely in recursion. The rule is as follows: *All aggregates which are defined without any* `freturn` *rule are monotonic and can be used freely in recursive rules.*

The ability of freely using aggregates with early returns in programs allows us to express concisely complex algorithms. For instance, we next define a continuous count that returns the current count after each new element but the first one (thus, it does not have a freturn since that would be redundant).

$$\begin{aligned}
&\texttt{single}(\texttt{mcount}, \texttt{Y}, 1).\\
&\texttt{multi}(\texttt{mcount}, \texttt{Y}, \texttt{Old}, \texttt{New}) \leftarrow \quad \texttt{New} = \texttt{Old} + 1.\\
&\texttt{ereturn}(\texttt{mcount}, \texttt{Y}, \texttt{Old}, \texttt{New}) \leftarrow \texttt{New} = \texttt{Old} + 1.
\end{aligned}$$

Using `mcount` we can now code the following applications, taken from [24].

Join the Party Some people will come to the party no matter what, and their names are stored in a `sure(Person)` relation. But others will join only after they know that at least $K = 3$ of their friends will be there. Here, `friend(P,F)` denotes that F is P's friend.

$$\begin{aligned}
&\texttt{willcome}(\texttt{P}) \leftarrow && \texttt{sure}(\texttt{P}).\\
&\texttt{willcome}(\texttt{P}) \leftarrow && \texttt{c_friends}(\texttt{P}, \texttt{K}), \texttt{K} \geq 3.\\
&\texttt{c_friends}(\texttt{P}, \texttt{mcount}\langle \texttt{F} \rangle) \leftarrow && \texttt{willcome}(\texttt{F}), \texttt{friend}(\texttt{P}, \texttt{F}).
\end{aligned}$$

Consider now a computation of these rules on the following database.

$$\begin{aligned}
&\texttt{friend}(\texttt{jerry}, \texttt{mark}). && \texttt{sure}(\texttt{mark}).\\
&\texttt{friend}(\texttt{penny}, \texttt{mark}). && \texttt{sure}(\texttt{tom}).\\
&\texttt{friend}(\texttt{jerry}, \texttt{jane}). && \texttt{sure}(\texttt{jane}).\\
&\texttt{friend}(\texttt{penny}, \texttt{jane}).\\
&\texttt{friend}(\texttt{jerry}, \texttt{penny}).\\
&\texttt{friend}(\texttt{penny}, \texttt{tom}).
\end{aligned}$$

Then, the basic semi-naive computation yields:

$$\texttt{willcome}(\texttt{mark}), \texttt{willcome}(\texttt{tom}), \texttt{willcome}(\texttt{jane}),$$
$$\texttt{c\_friends}(\texttt{jerry}, 1), \texttt{c\_friends}(\texttt{penny}, 1), \texttt{c\_friends}(\texttt{jerry}, 2),$$
$$\texttt{c\_friends}(\texttt{penny}, 2), \texttt{c\_friends}(\texttt{penny}, 3), \texttt{willcome}(\texttt{penny}),$$
$$\texttt{c\_friends}(\texttt{jerry}, 3), \texttt{willcome}(\texttt{jerry}).$$

This example illustrates how the standard semi-naive computation can be applied to queries containing monotonic user-defined aggregates. Another interesting example is transitive ownership and control of corporations [24].

Company Control Say that $\texttt{owns}(\texttt{C1}, \texttt{C2}, \texttt{Per})$ denotes the percentage of shares that corporation C1 owns of corporation C2. Then, C1 controls C2 if it owns more than, say, 50% of its shares. In general, to decide whether C1 controls C3 we must also add the shares owned by corporations such as C2 that are controlled by C1. This yields the transitive control rules defined with the help of a continuous sum aggregate that returns the partial sum for each new element, but the first one.

$$\texttt{control}(\texttt{C}, \texttt{C}) \leftarrow \quad \texttt{owns}(\texttt{C}, \_, \_).$$
$$\texttt{control}(\texttt{Onr}, \texttt{C}) \leftarrow \quad \texttt{twons}(\texttt{Onr}, \texttt{C}, \texttt{Per}), \texttt{Per} > 50.$$
$$\texttt{towns}(\texttt{Onr}, \texttt{C2}, \texttt{msum}\langle\texttt{Per}\rangle) \leftarrow \texttt{control}(\texttt{Onr}, \texttt{C1}), \texttt{owns}(\texttt{C1}, \texttt{C2}, \texttt{Per}).$$

Thus, every company controls itself, and a company C1 that has transitive ownership of more than 50% of C2's shares controls C2 . In the last rule, twons computes transitive ownership with the help of msum that adds up the shares of controlling companies. Observe that any pair (Onr, C2) is added at most once to control, thus the contribution of C1 to Onr's transitive ownership of C2 is only accounted once.

Bill-of-Materials (BoM) Applications BoM applications represent an important application area that requires aggregates in recursive rules. Say, for instance that $\texttt{assembly}(\texttt{P1}, \texttt{P2}, \texttt{QT})$ denotes that P1 contains part P2 in quantity QT. We also have elementary parts described by the relation $\texttt{basic\_part}(\texttt{Part}, \texttt{Price})$. Then, the following program computes the cost of a part as the sum of the cost of the basic parts it contains.

$$\texttt{part\_cost}(\texttt{Part}, 0, \texttt{Cst}) \leftarrow \quad \texttt{basic\_part}(\texttt{Part}, \texttt{Cst}).$$
$$\texttt{part\_cost}(\texttt{Part}, \texttt{mcount}\langle\texttt{Sb}\rangle, \texttt{msum}\langle\texttt{MCst}\rangle) \leftarrow$$
$$\texttt{part\_cost}(\texttt{Sb}, \texttt{ChC}, \texttt{Cst}), \texttt{prolfc}(\texttt{Sb}, \texttt{ChC}),$$
$$\texttt{assembly}(\texttt{Part}, \texttt{Sb}, \texttt{Mult}), \texttt{MCst} = \texttt{Cst} * \texttt{Mult}.$$

Thus, the key condition in the body of the second rule is that a subpart Sb is counted in part_cost only when all of Sb's children have been counted. This occurs when the number of Sb's children counted so far by mcount is equal to the out-degree of this node in the graph representing assembly. This number is kept in the prolificacy table, $\texttt{prolfc}(\texttt{Part}, \texttt{ChC})$, which can be computed as follows:

$$\texttt{prolfc}(\texttt{P1}, \texttt{count}\langle\texttt{P2}\rangle) \leftarrow \texttt{assembly}(\texttt{P1}, \texttt{P2}, \_).$$
$$\texttt{prolfc}(\texttt{P1}, 0) \leftarrow \quad \texttt{basic\_part}(\texttt{P1}, \_).$$

8 Conclusions

Keys in derived relations extend the expressive power of deductive databases while retaining their declarative semantics and efficient implementations. In this paper, we have presented equivalent fixpoint and model-theoretic semantics for programs with key constraints in derived relations. Database aggregates can be easily modelled under this extension, yielding a simple characterization of monotonic aggregates. Monotonic aggregates can be freely used in recursive programs, thus providing simple and efficient expressions for optimization and greedy algorithms that had been previously considered impervious to the logic programming paradigm.

There has been a significant amount of previous work that is relevant to the results presented in this paper. In particular the $\mathcal{LDL}$++ provides the choice construct to declare functional dependency constraints in derived relations. The stable model characterization and several other results presented in this paper find a similar counterpart in properties of $\mathcal{LDL}$++ choice construct [13,37]; however, no fixpoint characterization and related results were known for $\mathcal{LDL}$++ choice. An extension of this concept to temporal logic programming was proposed by Orgun and Wadge [21], who introduced the notion of choice predicates that ensure that a given predicate is single-valued. This notion finds applications in intensional logic programming [21].

The cardinality and weight constraints proposed by Niemelä and Simons provide a powerful generalization to key constraints discussed here [20]. In fact, while the key constraint restrict the cardinality of the results to be one, the constraint that such cardinality must be restricted within a user-specified interval is supported in the mentioned work (where different weights can also be attached to atoms). Thus Niemelä and Simons (i) provide a stable model characterization for logic programs containing such constraints, (ii) propose an implementation using Smodels [19], and (ii) show how to express NP-complete problems using these constraints. The implementation approach used for Smodels is quite different from that of $\mathcal{LDL}$++; thus investigating the performance of different approaches in supporting cardinality constraints represents an interesting topic for future research.

Also left for future research, there is the topic of SLD-resolution, which (along with the fixpoint and model-theoretic semantics treated here) would provide a third semantic characterization for logic programs with key constraints [29]. Memoing techniques could be used for this purpose, and for an efficient implementation of keys and aggregates [3].

Acknowledgements

The author would like to thank the reviewers for the many improvements they have suggested, and Frank Myers for his careful proofreading of the manuscript. The author would also like to express his gratitude to Dino Pedreschi, Domenico Saccá, Fosca Giannotti and Sergio Greco who laid the seeds of these ideas during our past collaborations.

This work was supported by NSF Grant IIS-007135.

References

1. S. Abiteboul, R. Hull, and V. Vianu: *Foundations of Databases*. Addison-Wesley, 1995.
2. N. Bidoit and C. Froidevaux: General logical Databases and Programs: Default Logic Semantics and Stratification. *Information and Computation*, 91, pp. 15–54, 1991.
3. W. Chen, D. S. Warren: Tabled Evaluation With Delaying for General Logic Programs. JACM, 43(1): 20-74 (1996).
4. D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S.Tsur and C. Zaniolo: The LDL System Prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), pp. 76-90, 1990.
5. S. Ceri, G. Gottlob and L. Tanca: *Logic Programming and Databases.* Springer, 1990.
6. S. W. Dietrich: Shortest Path by Approximation in Logic Programs. *ACM Letters on Programming Languages and Systems*, 1(2), pp. 119–137, 1992.
7. S. J. Finkelstein, N.Mattos, I.S. Mumick, and H. Pirahesh: Expressing Recursive Queries in SQL, ISO WG3 report X3H2-96-075, March 1996.
8. J. M. Hellerstein, P. J. Haas, H. J. Wang.: Online Aggregation. *SIGMOD 1997: Proc. ACM SIGMOD Int. Conference on Management of Data*, pp. 171-182, ACM, 1997.
9. S. Ganguly, S. Greco, and C. Zaniolo: Extrema Predicates in Deductive Databases. *JCSS* 51(2), pp. 244-259, 1995.
10. M. Gelfond and V. Lifschitz: The Stable Model Semantics for Logic Programming. *Proc. Joint International Conference and Symposium on Logic Programming*, R. A. Kowalski and K. A. Bowen (eds.), pp. 1070-1080, MIT Press, 1988.
11. F. Giannotti, D. Pedreschi, D. Saccà, C. Zaniolo: Non-Determinism in Deductive Databases. In *DOOD'91*, C. Delobel, M. Kifer, Y. Masunaga (eds.), pp. 129-146, Springer, 1991.
12. F. Giannotti, G. Manco, M. Nanni, D. Pedreschi: On the Effective Semantics of Nondeterministic, Nonmonotonic, Temporal Logic Databases. *Proceedings of 12th Int. Workshop, Computer Science Logic*, pp. 58-72, LNCS Vol. 1584, Springer, 1999.
13. F. Giannotti, D. Pedreschi, and C. Zaniolo: Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases. *JCSS* 62, pp. 15-42, 2001.
14. Sergio Greco, Domenico Saccà: NP Optimization Problems in Datalog. *ILPS 1997: Proc. Int. Logic Programming Symposium*, pp. 181-195, MIT Press, 1997.
15. S. Greco and C. Zaniolo: Greedy Algorithms in Datalog with Choice and Negation, *Proc. 1998 Joint Int. Conference & Symposium on Logic Programming*, JCSLP'98, pp. 294-309, MIT Press, 1998.
16. R. Krishnamurthy, S. Naqvi: Non-Deterministic Choice in Datalog. In *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, pp. 416-424, Morgan Kaufmann, 1988.
17. V. W. Marek and M. Truszczynski: *Nonmonotonic Logic.* Springer-Verlag, New York, 1995.
18. J. Minker: Logic and Databases: A 20 Year Retrospective. In D. Pedreschi and C. Zaniolo (eds.), *Proceedings International Workshop on Logic in Databases (LID'96)*, Springer-Verlag, pp. 5–52, 1996.
19. I. Niemelä, P. Simons and T. Syrjanen: Smodels: A System for Answer Set Programming Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, April 9-11, 2000, Breckenridge, Colorado, 4 pages. (Also see: http://www.tcs.hut.fi/Software/smodels/)

20. I. Niemelẗa and P. Simons: Extending the Smodels System with Cardinality and Weight Constraints. In Jack Minker (ed.): *Logic-Based Artificial Intelligence*, pp. 491-521. Kluwer Academic Publishers, 2001.
21. M.A. Orgun and W.W. Wadge,
Towards an Unified Theory of Intensional Logic Programming.
The Journal of Logic and Computation, 4(6), pp. 877-903, 1994.
22. T. C. Przymusinski: On the Declarative and Procedural Semantics of Stratified Deductive Databases: In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 193–216, Morgan Kaufmann, 1988.
23. R. Ramakrishnan, D. Srivastava, S. Sudanshan, and P. Seshadri: Implementation of the CORAL Deductive Database System. *SIGMOD'93: Proc. Int. ACM SIGMOD Conference on Management of Data*, pp. 167–176, ACM, 1993.
24. K. A. Ross and Yehoshua Sagiv: Monotonic Aggregation in Deductive Database, *JCSS* 54(1), pp. 79-97, 1997.
25. D. Saccà and C. Zaniolo: Deterministic and Non-deterministic Stable Models, *Journal of Logic and Computation*, 7(5), pp. 555-579, 1997.
26. J. S. Schlipf: Complexity and Undecidability Results in Logic Programming, *Annals of Mathematics and Artificial Intelligence*, 15, pp. 257-288, 1995.
27. S. Sudarshan and R. Ramakrishnan: Aggregation and relevance in deductive databases. *VLDB'91: Proceedings of 17th Conference on Very Large Data Bases*, pp. 501-511, Morgan Kaufmann, 1991.
28. J. D. Ullman: *Principles of Data and Knowledge-Based Systems*, Computer Science Press, New York, 1988.
29. M.H. Van Emden and R. Kowalski: The Semantics of Predicate Logic as a Programming Language. *JACM* 23(4), pp. 733-742, 1976.
30. A. Van Gelder, K. A. Ross, and J. S. Schlipf: The Well-Founded Semantics for General Logic Programs. *JACM* 38, pp. 620–650, 1991.
31. A. Van Gelder: Foundations of Aggregations in Deductive Databases. In *DOOD'93*, S. Ceri, K. Tanaka, S. Tsur (Eds.), pp. 13-34, Springer, 1993.
32. H. Wang and C. Zaniolo: User-Defined Aggregates in Object-Relational Database Systems. *ICDE 2000: International Conference on Database Engineering*. pp. 111-121, IEEE Press, 2000.
33. H. Wang and C. Zaniolo: Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. *VLDB 2000: Proceedings of 26th Conference on Very Large Data Bases*, pp. 166-175, Morgan Kaufmann, 2000.
34. C. Zaniolo and H. Wang: Logic-Based User-Defined Aggregates for the Next Generation of Database Systems. In K.R. Apt, V. Marek, M. Truszczynski, D.S. Warren (eds.): *The Logic Programming Paradigm: Current Trends and Future Directions.* Springer Verlag, pp. 121-140, 1999.
35. C. Zaniolo, S. Ceri, C. Faloutzos, R. Snodgrass, V.S. Subrahmanian, and R. Zicari: *Advanced Database Systems*, Morgan Kaufmann, 1997.
36. C. Zaniolo: The Nonmonotonic Semantics of Active Rules in Deductive Databases. In *DOOD 1997*, F. Bry, R. Ramakrishnan, K. Ramamohanarao (eds.), pp. 265-282, Springer, 1997.
37. C. Zaniolo et al.: $\mathcal{LDL}$++ Documentation and Web Demo, 1988: `http://www.cs.ucla.edu/ldl`
38. C. Zaniolo: Key Constraints and Monotonic Aggregates in Deductive Databases. UCLA technical report, June 2001.