Order in Datalog with Applications to Declarative Output

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
brass@informatik.uni-halle.de

Abstract. We propose an extension of Datalog that has "ordered predicates" (lists/arrays of tuples instead of sets of tuples). We previously suggested to specify output of Datalog programs declaratively by defining text pieces with their position. The proposal in the current paper reaches significantly farther by making order a first class citizen in the language. For database application programs, the output is an important part of the program, and should be fully integrated into the declarative language. However, order has many more applications besides specifying output. For instance, SQL has recently been extended by ranking functions, and aggregates over windows looping over sorted data — all this is needed in Datalog, too.

1 Introduction

Currently, database application programs are usually developed in a combination of two or more languages, e.g. PHP for programming and SQL for the database queries and updates. While SQL is declarative, most languages used for the programming part are not. However, SQL cannot be used for specifying complex output (e.g., generating a web page), so the programming part is necessary.

The goal of deductive databases is that a single, declarative language is used for programming *and* database tasks. The advantages of declarativity have been shown in SQL: The productivity is higher (because the programs are shorter and there is no need to think about efficient evaluation), and new technology (parallel hardware, new data structures/algorithms) can be used for existing application programs without changing them, because only the DBMS needs to be updated.

Although generating output is important in practice, it seems that there is no really good solution for Datalog yet. The standard solution in Prolog with a `write`-predicate is clearly non-declarative because it depends on the specific evaluation order used in Prolog. Non-declarative output might be acceptable for programs that do a complex calculation (specified declaratively), and contain only a small part that prints the result in the end. However, for database application programs, output is usually a significant part. Therefore, being able to specify the output declaratively is important in this case.

A well-known solution to declarative output in logic programming is to use a state argument as an accumulator pair (`IOStateIn`, `IOStateOut` in every predicate with output). While this is a good solution for programmers who think

"top-down" (if coupled with syntactic abbreviations and a determinism analysis as in Mercury [1]), it contradicts a basic requirement in deductive databases:

- In deductive databases, one usually thinks bottom-up. Thus, it should be possible to understand any kind of extended Datalog programs by applying (some variant of) the usual T_P (fixpoint) operator. I.e. a naive execution of the rules in the direction of the arrow "←" should be possible. Given this, using the state argument is simply no option: There could be infinitely many possible states for the first body literal of a rule. Furthermore, it seems natural that actions like output should be done in the head, not in the body.
- If Prolog is used for database applications, several solutions to a query are usually generated via backtracking (e.g. a relation is specified as a set of facts). But with the IO state solution, backtracking must be avoided (one cannot backtrack over an already performed output). Thus predicates like `findall` must be used, plus recursion over the produced list, even for really simple queries. This does not seem adequate and would deter users new to logic programming. Note that this is a consequence of the set-oriented evaluation which is a classical characteristic of deductive databases.

In the same way, using monads as in functional languages [7] is not a solution that fits to the classical Datalog bottom-up programming paradigm, and is therefore no alternative.

The basic idea of our approach is that predicates can be declared as ordered. This can be semantically understood as introducing an additional, usually hidden argument that defines the order of the facts for the predicates (of course, there are often more efficient implementations, which avoid actually storing such an argument).

If the order is not explicitly specified, the default for an ordered predicate takes into account the order of the facts or rules and the possible order defined for body literals (see Subsection 2.6). Thus, a sequence of facts becomes actually a list or array. Experience shows that relational tables not seldom need extra "position" columns for defining an order. Making this automatic and hidden could simplify working with such data.

As in [2], output is done by defining a predicate `output` which contains text pieces to be printed. With the ordered predicates of the current proposal, the position information becomes implicit, and a simple example is:

```
ordered output/1.
output('Hello, ').
output(Name) ← name(Name).
output('.\n').
name('Nina').
```

Of course, with special syntactic abbreviations, output can be made still more natural and easy (see the pattern syntax in Section 3).

But order has many more applications than just output, for instance

- ranking, top-n queries and window functions as recently added to SQL,

– list and array processing,
– aggregation functions, and
– specification of algorithms that pass through a sequence of computation states (i.e., more or less imperative algorithms).

The last two points appear already in LDL (XY-stratification) [8], but otherwise our approach is quite different and in some aspects more general.

Whereas in [2], all ordering requirements for facts of predicates were only implicitly derived from the requested ordering from the top `output` predicate, now explicit sorting information can be specified for any predicate. This facilitates the design of reusable components, and it also corresponds to a recent development in SQL. In SQL, top-n and window queries have recently become important — every major DBMS has special support for them, although the syntax is different in different systems. For instance, in Oracle it is possible to retrieve the three employees with highest salary as follows:

```
SELECT ENAME, SAL
FROM   (SELECT ENAME, SAL
        FROM   EMP
        ORDER  BY SAL DESC)
WHERE  ROWNUM <= 3
```

This example is interesting, because it shows that a subquery, corresponding to a view or a derived predicate, might need a defined order. In older SQL standards, it was not possible to use `ORDER BY` in subqueries or view definitions. In our proposal, the subquery corresponds to the following ordered predicate:

```
ordered emp_by_sal/2.
emp_by_sal<^Sal>(EName, Sal) ← emp(EName, Sal, Job).
```

In this case, the special ordering argument must be explicitly defined — this is done in "`<...>`". The "`^`" means "descending order", i.e. the highest salary first (the intuition is to think of "↑" instead of the default order "↓").

Now a powerful function of the system is that it can condense any ordering argument (possibly list-valued for several ordering criteria of different priority) to a single integer (in an order-preserving manner). There are several methods for doing this, but the default gives a simple array index:

```
answer(EName, Sal) ← emp_by_sal[N](EName, Sal) ∧ N ≤ 3.
```

This syntax corresponds to the understanding that `emp_by_sal` is now an array, and the array entries are records/facts.

The possibility to explicitly refer e.g. to the first, previous, next, and last fact with respect to some order significantly increases the expressiveness of the language. Of course, the construct is nonmonotonic, and already a check for the first tuple gives the possibility to simulate negation.

2 Datalog with Ordering: Syntax and Semantics

We start with Datalog with stratified negation. Terms are constants or variables (no function symbols). Rules must be range-restricted, i.e. variables that appear in the head literal or in a negative body literal must also appear in a positive body literal. Of course, all this could be generalized, but those are questions orthogonal to the issue of the current paper.

2.1 Declaration of Ordered Predicates

We assume that a subset of the predicates are declared as "ordered predicates":

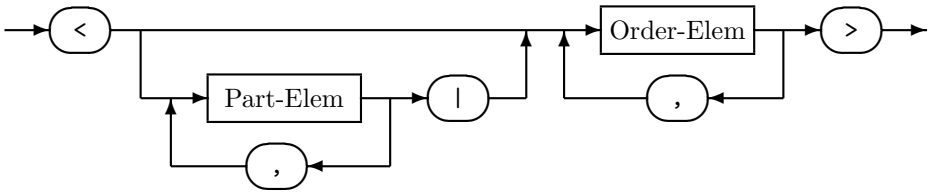
```
ordered p/n.
```

This makes predicate `p` of arity `n` an ordered predicate. When rules for `p` contain explicit ordering specifications, a special declaration is not necessary. However, there is also a default sort order for ordered predicates, and then it must be made clear that this is not a normal predicate.

2.2 Order Specifications in the Head Literal

If the predicate in the head literal is an ordered predicate, an order specification is expected after the predicate name and before the argument list. The order specification is written in angle brackets `<...>`. It consists of an optional partitioning part, and an ordering part. Partitioning is necessary if one wants to rank the data values in several groups, e.g. the top 3 salaries for each job. This means that the order on the facts is not a linear order, but there can be incomparable facts. One can also view the predicate as a two-dimensional array, where the first dimension is the partitioning value (e.g. the job), and the second is the position in the defined order. With partitioning, it is possible to represent more than one list in a predicate.

Order-Specification:



The partitioning part is a comma-separated list of partitioning elements, which are simply terms. Two facts are comparable if they agree in the values for all partitioning elements. Therefore, the sequence of the partitioning elements is not important.

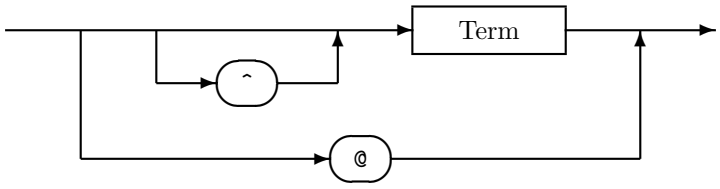
Part-Elem:

The ordering part is a comma-separated list of order elements, which are

- terms, possibly marked with the "descending" operator `^` (inversing the sort direction), or
- the special marker "`@`", which is replaced by the number of the current rule (this permits to order facts by in the sequence in which they are written in the program without having to use explicit numeric "labels" as in Basic).

The value of the first ordering element has highest priority in determining the sort order of two facts, and only if it is equal, the value of the second ordering element is considered, and so on (as in the `ORDER BY` clause of SQL).

Order-Elem:



2.3 Accessing the Sort Index in Body Literals

For body literals with an ordered predicate, one can optionally access the position in the ordered list (the "array index"). However, if several facts have the same ordering value (e.g. several employees with equal salary), this position (the `ROWNUM` or `ROW_NUMBER` in SQL), is arbitrary (defined by the implementation). Therefore, SQL introduced two more ranking functions:

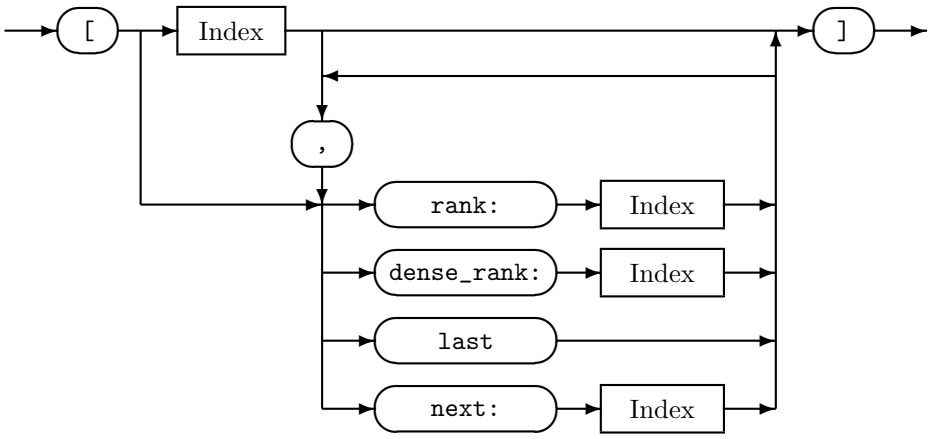
- `RANK` counts the number of tuples/facts with an ordering value less than the value in the current fact/tuple (and then adds 1, so that the first position is 1 and not 0).
- `DENSE_RANK` counts the number of distinct ordering values less than the value in the current fact/tuple (again, 1 is added).

We also permit to check for the last tuple and to get the index of the next tuple (with respect to the row number, i.e. the standard array index):

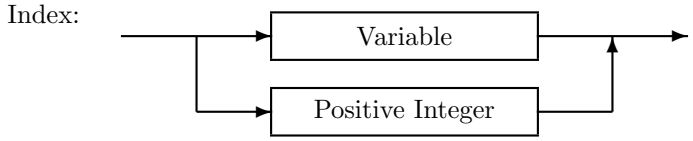
EName	Sal	row_number	rank	dense_rank	last	next
Andrew	4000	1	1	1	false	2
Betty	3000	2	2	2	false	3
Chris	3000	3	2	2	false	4
Doris	2000	4	4	3	false	5
Eddy	1000	5	5	4	false	6
Fred	1000	6	5	4	true	nil

All these functions can be accessed in a single pair of brackets, since these functions can be determined in a single scan over the sorted list. The functions are distinguished by a prefix in front of the index, only the row number has no prefix, because it is most similar to an array index.

Order-Position:



An index is a variable or a positive integer constant:



As an example, consider the following SQL query:

```
SELECT ENAME, SAL, RANK() OVER (ORDER BY SAL DESC)
FROM   EMP
ORDER  BY ENAME
```

This shows that more than one ordering might be necessary during the evaluation of a query: First, the employee tuples must be sorted by salary in order to compute the rank (position in that order), and then the tuples must be sorted by employee name for output. This looks as follows in our Datalog extension:

```
emp_by_sal<^Sal>(EName, Sal) ← emp(EName, Sal, Job).
answer<EName>(EName, Sal, N) ← emp_by_sal[rank:N](EName, Sal).
```

Later, we will discuss a possibility to use an order specification directly in the body literal, so that the auxillary predicate `emp_by_sal` can be avoided.

2.4 Stratification

It is not surprising that with recursion and the possibility to determine the first literal, one can get contradictory/inconsistent situations:

```
p<10>(a) ← p[1](b).
p<20>(b).
```

If `p(b)` is the first element in the sorted list `p`, then `p(a)` is true, which then would come first. But then `p(b)` is no longer the first element. This program has

no reasonable semantics and must be excluded. The solution is the same as in the case of negation: We require that there is a level mapping l, which assigns positive integers to the predicates, such that for rules containing `p[...](...)` in the body, and `q` in the head, $l(\mathtt{q}) > l(\mathtt{p})$. In any case, if `p` occurs in the body, and `q` in the head, $l(\mathtt{q}) \geq l(\mathtt{p})$.

The stratification ensures that it is possible to compute all facts about an ordered predicate before the position of a fact can be determined. It is possible to reduce the stratification requirements at least in the following cases:

- When the index position in the body is only used to determine an ordering value in the head, the exact value is not important if this is the only rule about the predicate, or the rule number is a sort criterion of higher priority. Then any order-preserving values can be used, and recursion is possible.
- When it can be ensured that a recursive rule yields only facts that will get a higher index than facts used in the body, also no contradiction can occur (this is similar to XY-stratification in LDL).

However, in order to keep the explanations simple, we will assume the stratification requirement in the following.

2.5 Evaluation

We propose a simple, "naive" evaluation method here in order to define clearly the semantics of programs in our extended Datalog dialect. Of course, for real query evaluation, many optimizations are possible (and needed in order to reach acceptable performance). But space is not sufficient to treat this topic here.

Our approach consists of rewriting the rules, evaluating them in the order of the stratification levels bottom-up, and having a special sorting and ranking step between the fixpoint computations for each level. For every ordered predicate `p` of arity n, two new predicates are introduced:

- `p_head` of arity $n + 2$, where the additional arguments are used for the partitioning and ordering values (stored as lists),
- `p_body` of arity $n + 4$, where the additional values are used for `row_number` (normal index), `rank`, `dense_rank`, and `next`.

The rules are rewritten in the obvious way: The head literal

$$\mathsf{p}\langle \mathsf{a}_1, \ldots, \mathsf{a}_k | \mathsf{b}_1, \ldots, \mathsf{b}_m \rangle (\mathsf{t}_1, \ldots, \mathsf{t}_n)$$

is translated to

$$\mathsf{p\_head}([\mathsf{a}_1, \ldots, \mathsf{a}_k],\ [\bar{\mathsf{b}}_1, \ldots, \bar{\mathsf{b}}_m],\ \mathsf{t}_1, \ldots, \mathsf{t}_n),$$

where $\bar{\mathsf{b}}_i$ is

- $\mathsf{desc}(\mathsf{b}'_i)$ if b_i has the form $\hat{}\,\mathsf{b}'_i$,
- the number of the current rule, if b_i is "`@`", and
- b_i otherwise.

The body literal

$$\mathsf{p}[\mathsf{a}_1,\ \mathsf{rank}\colon \mathsf{a}_2,\ \mathsf{dense\_rank}\colon \mathsf{a}_3,\ \mathsf{next}\colon \mathsf{a}_4](\mathsf{t}_1, \ldots, \mathsf{t}_n)$$

is translated to

$$\mathsf{p\_body}(\mathsf{a}_1,\ \mathsf{a}_2,\ \mathsf{a}_3,\ \mathsf{a}_4,\ \mathsf{t}_1, \ldots, \mathsf{t}_n),$$

where for each part that is not used, an anonymous variable a_i is inserted. If `last` appears in the index, $\mathsf{a}_4 = \mathsf{nil}$ is added to the body of the rule (or nil is directly inserted in the fourth argument position if next is not used).

Now the following algorithm can be used for query evaluation:

```
(1)  Let the input program P be split into strata P_1, ..., P_m;
(2)  F := ∅; /* set of derived facts to be computed */
(3)  for i = 1, ..., m {
(4)      Let P'_i be the rewritten version of P_i;
(5)      Compute the fixpoint F_i of T_{P'_i ∪ F};
(6)      F := F ∪ {p(c_1, ..., c_n) ∈ F_i | p is standard predicate of level i};
(7)      foreach ordered predicate p of stratification level i {
(8)          Sort the facts about p_head in F_i by first arg., second arg.
(9)          last_a := nil; last_b := nil; last_fact := nil;
(10)         foreach fact p_head(a, b, c_1, ..., c_n) in sorted sequence {
(11)             if (a ≠ last_a) { /* new partition */
(12)                 rownum := 1; rank := 1; dense_rank := 1;
(13)                 if (last_fact ≠ nil) insert last_fact into F;
(14)             } else { /* Not first row in partition */
(15)                 rownum++;
(16)                 if (b ≠ last_b) {
(17)                     rank = rownum;
(18)                     dense_rank++;
(19)                 }
(20)                 insert last_fact into F with 4th arg (nil)
(21)                                     replaced by rownum;
(22)             }
(23)             last_a := a; last_b := b;
(24)             last_fact := p_body(rownum, rank, dense_rank, nil,
(25)                                     c_1, ..., c_n);
(26)         }
(27)         if (last_fact ≠ nil) insert last_fact into F;
(28)     }
(29) }
(30) Print tuples in F of main predicate (e.g. output_body or answer_body)
(31)     without the first four added arguments, sorted by first arg.
```

The sorting requires some explanation. For the partitioning argument, no particular sort sequence is needed. It is only important that facts with the same value in the partitioning argument appear next to each other. If the partitioning argument is equal, the order argument determines the sort sequence. The order argument is a list, and the first position, where the two lists differ, determines the order of the two facts. If one list is a prefix of the other, the shorter list comes first. Otherwise, the list elements are ordered as follows:

– numbers come first (in the usual order),
– then strings (in alphabetic order),
– then terms `desc(S)`, where `S` is a string (in inverse alphabetic order),
– then terms `desc(N)`, where `N` is a number (in inverse numeric order).

If different types are compared, this is likely an error, and one could print a warning. It is possible that distinct facts have identical ordering values: For the functions `rank` and `dense_rank` this is important, for the function `row_number` (and thus for output) the implementation can decide which fact to put first.

2.6 An Abbreviation: Default Order Specification

If a predicate is declared as ordered, but the head of a rule contains no order specification, a default order specification is constructed as follows: First the number of the rule, and then the index value of every body literal with an ordered predicate in the body (in the order of occurrence in the body). E.g. consider

$$\texttt{p(...)} \leftarrow \texttt{q(...)} \wedge \texttt{r(...)} \wedge \texttt{s(...).}$$

If this is the i-th rule about `p`, and `p`, `q`, `s` are ordered predicates (while `r` is a standard predicate), this rule is an abbreviation for

$$\texttt{p<i,X,Y>(...)} \leftarrow \texttt{q[X](...)} \wedge \texttt{r(...)} \wedge \texttt{s[Y](...).}$$

Here `X` and `Y` are the index positions of the facts used for the body literals with ordered predicates — this order is reflected in the derived facts.

Note that this order corresponds to the order in which Prolog would compute the `p`-facts (for non-recursive rules). Note also that rules become somewhat similar to comprehension syntax [3]: One constructs a list in the body with ordered predicates and can then add other conditions to filter the list elements.

If an ordered predicate is defined by a set of facts, without explicit order specification, the default order is the order in which the facts are written down.

This abbreviation also fits to the understanding of a query (goal) as a rule body: It suffices to translate

$$\leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_n \wedge \mathsf{L}_{n+1} \wedge \cdots \wedge \mathsf{L}_m.$$

to

$$\texttt{ordered answer}/k.$$
$$\texttt{answer}(X_1, \ldots, X_k) \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_n \wedge \mathsf{L}_{n+1} \wedge \cdots \wedge \mathsf{L}_m.$$

where $X_1, \ldots, X_k$ are the variables appearing in the query ("answer variables"). Then the default order specification is used to automatically propagate orders declared for the query literals to the result of the query.

3 Applications to Output

As mentioned in the introduction, output is done by defining an ordered predicate "`output`" which contains text pieces to be printed in the defined order. Of course, the predicate "`output`" is only the "main" predicate, which composes the final document out of text fragments defined in many other ordered predicates.

Suppose we want to generate an HTML-table with name and salary of the employees, ordered first by salary (descending) and for equal salary by name. Using standard rules with ordered predicates this is possible, but for longer texts, it looks somewhat clumsy. Instead, we propose an "output pattern" syntax, in which one can write any text (between the special symbols "`(#`" and "`#)`"), and mark places in the text where argument values "`<$>`" or texts defined by other predicates "`<#>`" should be inserted. Each pattern corresponds to a predicate.

```
sal_table(#
    <table>
    <tr><th>Employee</th><th>Salary</th></tr>
    <#sal_table_row>
    </table>
#).

sal_table_row<^Sal,EName>(#
        <tr><td><$EName></td><td><$Sal></td></tr>
#) ← emp(EName, Sal, _).
```

This is automatically translated to standard rules with ordered predicates by splitting the text of the pattern into pieces where something must be inserted. Note that the piece numbers do not have to be written by the user, they are automatically assigned by the system. Even if one writes the rules directly, piece numbers can usually be avoided or replaced by "`@`" (current rule number).

```
sal_table<1>('<table>\n').
sal_table<2>('<tr><th>Employee</th><th>Salary</th></tr>\n').
sal_table<3,Pos>(Text) ← sal_table_row[Pos](Text).
sal_table<4>('\n</table>\n').

sal_table_row<^Sal, EName, 1>('<tr><td>')   ← emp(Ename, Sal, _).
sal_table_row<^Sal, EName, 2>(EName)        ← emp(Ename, Sal, _).
sal_table_row<^Sal, EName, 3>('</td><td>')  ← emp(Ename, Sal, _).
sal_table_row<^Sal, EName, 4>(Sal)          ← emp(Ename, Sal, _).
sal_table_row<^Sal, EName, 5>('</td></tr>') ← emp(Ename, Sal, _).
```

4 More Applications

The possibility to number facts and to compute the next number permits to write loops over sets of facts. This can be used to write aggregation functions. E.g., the following program computes the sum of all salaries.

```
emp_list<EName>(EName, Sal) ← emp(EName, Sal, Job).
sal_sum(1, 0).
sal_sum(N1, S1) ←
    sal_sum(N, S),
    emp_list[N,next:N1](EName, Sal),
    S1 is S + Sal.
answer(S) ← sal_sum(nil, S).
```

5 Discussion of Alternatives

Quite often, the values used for ordering and partitioning are arguments of the head literal, so they could be directly marked there: `emp_list(EName, <^Sal>)`. It would also be possible to declare sorting and partitioning in the "`ordered`"-declaration for the predicate. However, this works only sometimes: E.g. for output applications, there often is no explicit ordering argument, and each rule has a different ordering specification. The solution proposed here is more general.

Instead of a single declaration "`ordered`", one could consider many different types of predicates. For instance, an "`ordered_set`" might eliminate duplicate facts which differ only in the value of the hidden ordering argument (one could use always the smallest/first ordering value) — this would be helpful for termination if one allows recursion. The converse case is when the hidden argument is used only for duplicates, and no ordering is required.

User-defined orders could be permitted. Besides several chains of linear orders, as in the current proposal, arbitrary partial orders could be used.

If one wants to determine the sequence number of a row, it looks nice if the ordering specification can be done in the body literal, i.e. the top-earning employees could be determined without an auxillary predicate:

$$\texttt{answer(EName, Sal)} \leftarrow \texttt{emp<^Sal>[N](EName, Sal, Job)} \wedge \texttt{N} \leq 3.$$

However, consider this case:

$$\begin{aligned}&\texttt{answer(EName, Sal)} \leftarrow \texttt{emp<^Sal>[N](EName, Sal, Job)} \wedge \\ &\qquad\qquad\qquad\qquad \texttt{N} \leq 3 \wedge \texttt{programmer(Job).} \\ &\texttt{programmer('Programmer').}\end{aligned}$$

The question is whether only programmers are considered when assigning row numbers, or row numbers are assigned first, and then programmers are selected (in which case the result may be empty when there is no programmer among the top earning employees). The problem is that it is no longer sufficient to consider a single assignment of values to variables when the rule is applied. Basically an aggregation is done here in the body (one counts the number of higher earning employees). The predicate `findall` in Prolog has a similar problem ($\wedge$ is not commutative). A solution is to mark arguments that should not be used for the purpose of determining row numbers, i.e. that are temporarily replaced by an anonymous variable, and after the numbers are assigned, the original argument is used (which can perform a selection or join). This can also be described by introducing a temporary predicate for the call.

6 Related Work

Datalog with arrays was studied in [5, 4], but there the arrays are terms and the emphasis is on using the indexed memory access for efficiency. Datalog with a multiset semantics for predicates (i.e. allowing duplicates) was investigated in [6]. They use "colored sets" which is similar to an additional hidden argument. A database query language for more general collection types based on comprehension syntax was discussed in [3].

7 Conclusions

We proposed an extension of Datalog that permits to specify the order of facts for predicates. If the vision of a deductive database as a declarative, integrated system for developing database applications should come true, such an extension is needed for two reasons: (1) The system must support more or less all features of SQL, and SQL has not only `ORDER BY`, but also functions like `RANK`, which permit to use order in conditions. (2) Output is an essential part of database applications, and output is necessarily a sequence of text pieces. Actually, every query result that consists of more than just a few rows will be easier to read and understand if it is ordered in some reasonable way.

The proposed extension is also interesting for the following reasons: (3) It permits to work on lists in a very direct and simple way, without structured terms, and often without recursion. This might help the non-sophisticated user. (4) For the power user, also loops over facts can be written, e.g. for computing aggregation functions. In this way, the expressiveness of the language is extended.

A small prototype is being implemented that allows to experiment with the language, see

`http://www.informatik.uni-halle.de/~brass/order/`

The prototype is also interesting because the input program is internally represented as a set of Datalog facts. Our long-term goal is to develop a Datalog-to-C++ compiler written in Datalog. The constructs introduced in this paper, especially for output, are necessary for this purpose.

References

1. Becket, R.: Mercury tutorial. Tech. rep., University of Melbourne, Dept. of Computer Science (2010)
2. Brass, S.: Declarative output by ordering text pieces. In: Gallagher, J., Gelfond, M. (eds.) Technical Communications of the 27th International Conference on Logic Programming (ICLP'11). Leibniz International Proceedings in Informatics (LIPIcs), vol. 11, pp. 151–161. Schloss Dagstuhl (2011)
3. Buneman, P., Libkin, L., Suciu, D., Tannen, V., Wong, L.: Comprehension syntax. SIGMOD Record 23(1), 87–96 (1994)
4. Greco, S., Palopoli, L., Spadafora, E.: Querying datalog with arrays: Design and implementation issues. Journal of Systems Integration 6, 299–327 (1996)
5. Greco, S., Palopoli, L., Spadafora, E.: Extending datalog with arrays. Data & Knowledge Engineering 17(1), 31–57 (October 1995)
6. Mumick, I.S., Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates. In: McLeod, D., Sacks-Davis, R., Schek, H.J. (eds.) Proc. of the 16th International Conf. on Very Large Data Bases (VLDB'90). pp. 264–277. Morgan Kaufmann (1990)
7. Wadler, P.: How to declare an imperative. ACM Computing Surveys 29(3), 240–263 (1997), see also: `http://homepages.inf.ed.ac.uk/wadler/topics/monads.html`
8. Zaniolo, C., Arni, N., Ong, K.: Negation and aggregates in recursive rules: The LDL++ approach. In: Ceri, S., Tanaka, K., Tsur, S. (eds.) Proceedings of the 3rd International Conference on Deductive and Object-Oriented Databases (DOOD'93). pp. 204–221. No. 760 in LNCS, Springer (1993)