Order in Datalog with Applications to Declarative Output

Stefan Brass
University of Halle, Germany

Overview

1. Motivation: Output, Ordered Predicates

2. Motivation: SQL, Ranking

3. Semantics

4. Aggregation (short)

5. Conclusions

Motivation (1)

A deductive database is . . .

- not only a system permitting recursive queries,

  That turned out to be no “quantum leap”.

- but a platform for developing database applications using a declarative language for database queries and programming (Datalog).

  SQL is declarative, but lacks the programming part. Therefore, database applications are developed today using a mixture of languages, e.g. a combination with PHP or other non-declarative languages.

Motivation (2)

- Output is an essential part of many database applications. It should be done declaratively.

  In this way they differ from programs that do a complicated computation and then print a short result. For such programs, a non-declarative solution for output might be ok. For database applications, it is not.

- In Datalog it is natural to understand the rules applied from body to head ($\sim$ bottom-up evaluation).

  Therefore actions, such as output, should be done in the head.

- Database relations are specified as a set of facts. Printing an entire relation should be a simple task without `findall` to avoid backtracking over output.

Ordered Predicates (1)

- In any programming language, output is done by constructing a sequence of text pieces.
- We use “ordered predicates”, which have an additional argument defining the order (written <...>).

```
ordered output/1.
output<1>('Hello, ').
output<2>(Name) ← name(Name).
output<3>('!\n').
name(Nina').
```

- Since the default value for the special argument is the rule number, it can be left out in the example.

Ordered Predicates (2)

- The ordering argument is list-valued to support several sorting criteria of different priority.
- In this way, also a tree structure of the document can easily be defined:

```
ordered output/1.
output<1>('<ul>\n').
output<2,Name,1>('<li>')     ←  programmer(Name).
output<2,Name,2>(Name)       ←  programmer(Name).
output<2,Name,3>('</li>\n')  ←  programmer(Name).
output<3>('</ul>\n').

programmer(Name) ← emp(Name, 'Programmer', Sal).
```

Ordered Predicates (3)

- Not only the "main predicate" output is ordered, but one can use auxillary ordered predicates:

```
ordered output/1, list_body/2, list_item/2.

output('<ul>\n').
output(Text) ← list_body(Text).
output('</ul>\n').

list_body<Name,i>(Text) ← list_item[i](Name, Text).

list_item(Name, '<li>')  ← programmer(Name).
list_item(Name, Name)    ← programmer(Name).
list_item(Name, '</li>') ← programmer(Name).
```

- Uses default order, except for sorting by name.

Ordered Predicates (4)

- In the rule body, one can access the position of the fact currently matched with the body literal:

```
list_body<Name,i>(Text) ← list_item[i](Name, Text).
```

- So the systems sorts the derivable facts and then assigns array indexes.

  The original list-valued ordering argument cannot be accessed in the body. We try to make it unnecessary to construct the list explicitly.

- The default order specification consists of the rule number, followed by the index positions of all body literals with ordered predicates in the order of appearance in the body (∼ Prolog computation).

Note

- Of course, the additional argument is only an easy way to explain the semantics.
- The syntax must be such that
  - ◇ it is usually not necessary to write the ordering argument explicitly (especially no numbers),
  - ◇ larger portions of text can be written as they will be printed (with markers for insertion places).
- Query evaluation should often be possible without explicit construction of the ordering argument.
- We have (preliminary) solutions for both problems.

Pattern Syntax for Output

- Permits a block of text to be written as it will appear in the output:

```
output(#
    <ul>
    <#list_body>
    </ul>
#).

list_body<Name>(#
    <li><$Name><li>
#) ← programmer(Name).
```

- Automatically translated into standard rules.

Overview

1. Motivation: Output, Ordered Predicates
2. Motivation: SQL, Ranking
3. Semantics
4. Aggregation (short)
5. Conclusions

Simple SQL Query in Datalog

- E.g. employees ordered by salary (highest first):

```
SELECT ENAME, SAL
FROM   EMP
ORDER  BY SAL DESC
```

- Same Query in Datalog with ordered predicates:

```
answer<^Sal>(EName, Sal) ← emp(EName, Sal, Job).
```

  `^Sal` is an abbreviation for `desc(Sal)`.

- The system has two possible main predicates:
  - ◇ `output/1`: Simple concatenation of text pieces.
  - ◇ `answer/n`: Produces tabular output.

Motivation: SQL

- Because of top-N, ranking and window queries,
  - ◇ order is also important semantically for the query result itself,
  - ◇ not only something cosmetic needed only at the end for printing.

    These constructs were recently added to SQL.
    From SQL-2003 to SQL-2008, the `ORDER BY` clause was added to view definitions (corresponding to derived predicates).
- Many different orders can be needed in one query.
- A deductive database system will not be successful if it does not permit an easy transition from SQL.

Example (1)

- E.g. jobs of the five employees with highest salary:

```
SELECT DISTINCT JOB
FROM   (SELECT JOB, ROW_NUMBER() OVER
                    (ORDER BY SAL DESC) N
        FROM   EMP)
WHERE  N <= 5
ORDER  BY JOB
```

- This query needs to sort the data two times:
  - ◇ First by salary to compute the position `N`,
  - ◇ then by job to produce the sorted output.

Example (2)

- Define a list/array of employee tuples ordered by descending salary:

```
        ordered emp_by_sal/3.
        emp_by_sal<^Sal>(EName, Job, Sal) ←
            emp(EName, Job, Sal).
```

- The system orders the derived facts by the special argument and assigns positions (row numbers):

```
    ordered answer/1.
    answer<Job>(Job) ←
        emp_by_sal[N](EName, Job, Sal) ∧ N ≤ 5.
```

- For equal salaries: implementation chooses order.

Ranking Functions

- In order to avoid the implementation-dependency, different ranking functions can be used as in SQL:

EName	Sal	row_number	rank	dense_rank
Andrew	4000	1	1	1
Betty	3000	2	2	2
Chris	3000	3	2	2
Doris	2000	4	4	3
Eddy	1000	5	5	4
Fred	1000	6	5	4
Gerd	800	7	7	5

```
answer<Job>(Job) ←
    emp_by_sal[rank:N](EName, Job, Sal) ∧ N ≤ 5.
```

Partitioning

- Sometimes, the row numbers or ranks are not needed for the entire predicate, but for a group of facts with certain equal arguments (∼ multidim. array).

  If one wants to pass bindings as in the magic set method, this is helpful (of course, if the concrete index values are not needed, but only there relative order, one can also avoid computing the entire extension).

- E.g. top 3 earning employees for each job:

```
job_emp<Job|^Sal>(EName, Job, Sal) ←
    emp(EName, Job, Sal).
answer<Job,N>(Job, N, EName) ←
    job_emp[N](EName, Job, Sal) ∧ N ≤ 3.
```

Overview

1. Motivation: Output, Ordered Predicates

2. Motivation: SQL, Ranking

3. Semantics

4. Aggregation (short)

5. Conclusions

Stratification (1)

- Negation can be simulated, e.g. by adding a dummy element which will certainly be last and checking whether it is also first.
- Therefore it is no surprise that one can write meaningless programs (with “odd loops”):

```
ordered p/1.
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 would then come first. But then p(b) is no longer the first element.

Stratification (2)

- The solution is the same as for negation:
  - ◇ Recursion through the determination of a row number or rank is forbidden, i.e.
  - ◇ it must be possible to assign levels to predicates such that the level of the predicate in the head is strictly greater than the level of a predicate used with `[...]` in the body

    (and the same for a predicate used negated in the body), and of course the level of the predicate in the head must always be greater than or equal to all predicate levels in the body.
    Thus the extension of a predicate can be fully computed before row numbers must be assigned.

Semantics

- The rules are translated to standard Datalog with two special predicates for each ordered predicate `p`:
  - ◇ `p_head` has two additional list-valued arguments for the partitioning and for the ordering value.
  - ◇ `p_body` has four additional arguments, for row number, rank, dense rank, and next row number.
- For each stratification level i:
  - ◇ Standard fixpoint computation is done.
  - ◇ For all ordered predicates `p` of level i, the derived `p_head` facts are sorted, and the corresponding `p_body` facts are computed.

Overview

1. Motivation: Output, Ordered Predicates

2. Motivation: SQL, Ranking

3. Semantics

4. Aggregation (short)

5. Conclusions

Computation of Aggregations

- The automatic numbering of facts permits to loop over them and e.g. compute 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).
```

Of course, standard aggregation functions should be directly supported in the syntax, but this example is interesting for questions of expressive power. Similar to solution in $\mathcal{LDL}$ (XY-stratification).

Overview

1. Motivation: Output, Ordered Predicates

2. Motivation: SQL, Ranking

3. Semantics

4. Aggregation (short)

5. Conclusions

Conclusions (1)

- My goal is to develop a deductive DBS that supports also programming, not only database queries.
- The plan is to do this by translation from Datalog to C++.
- The deductive database system should be written itself in Datalog.

  Or at least an essential part of the system.
- Output is needed for this task.

Conclusions (2)

- It seems obvious to me that more or less all features of SQL must be supported in a deductive DBS that aims at practical usage.
- Features shown here for `ORDER BY` and ranking are needed for this.

  Duplicates can be handled with the additional argument, too.
- Small prototype to try out the language:

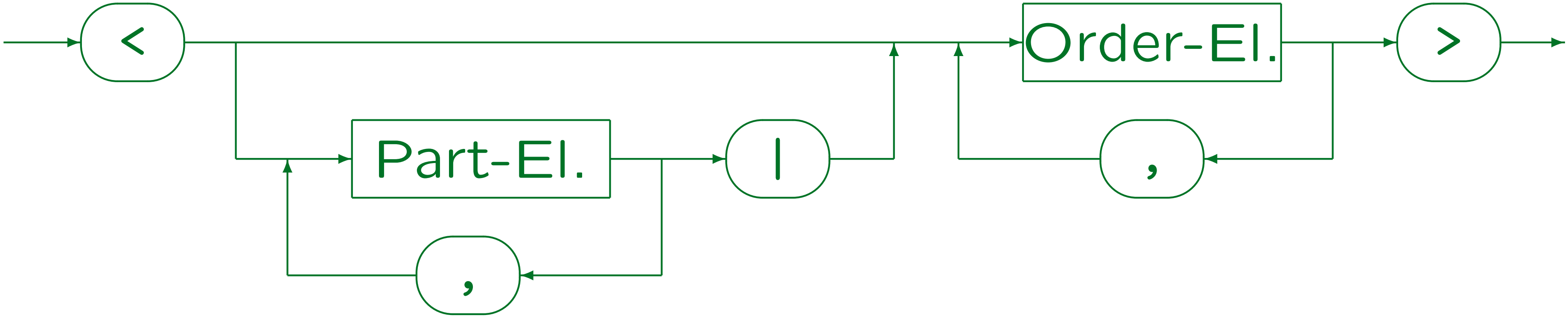
  http://www.informatik.uni-halle.de/~brass/order/
- The task is important. I made a proposal. Discussion on language syntax & semantics is welcome.

Overview

1. Motivation: Output, Ordered Predicates
2. Motivation: SQL, Ranking
3. Semantics
4. Aggregation (short)
5. Conclusions
6. Appendix

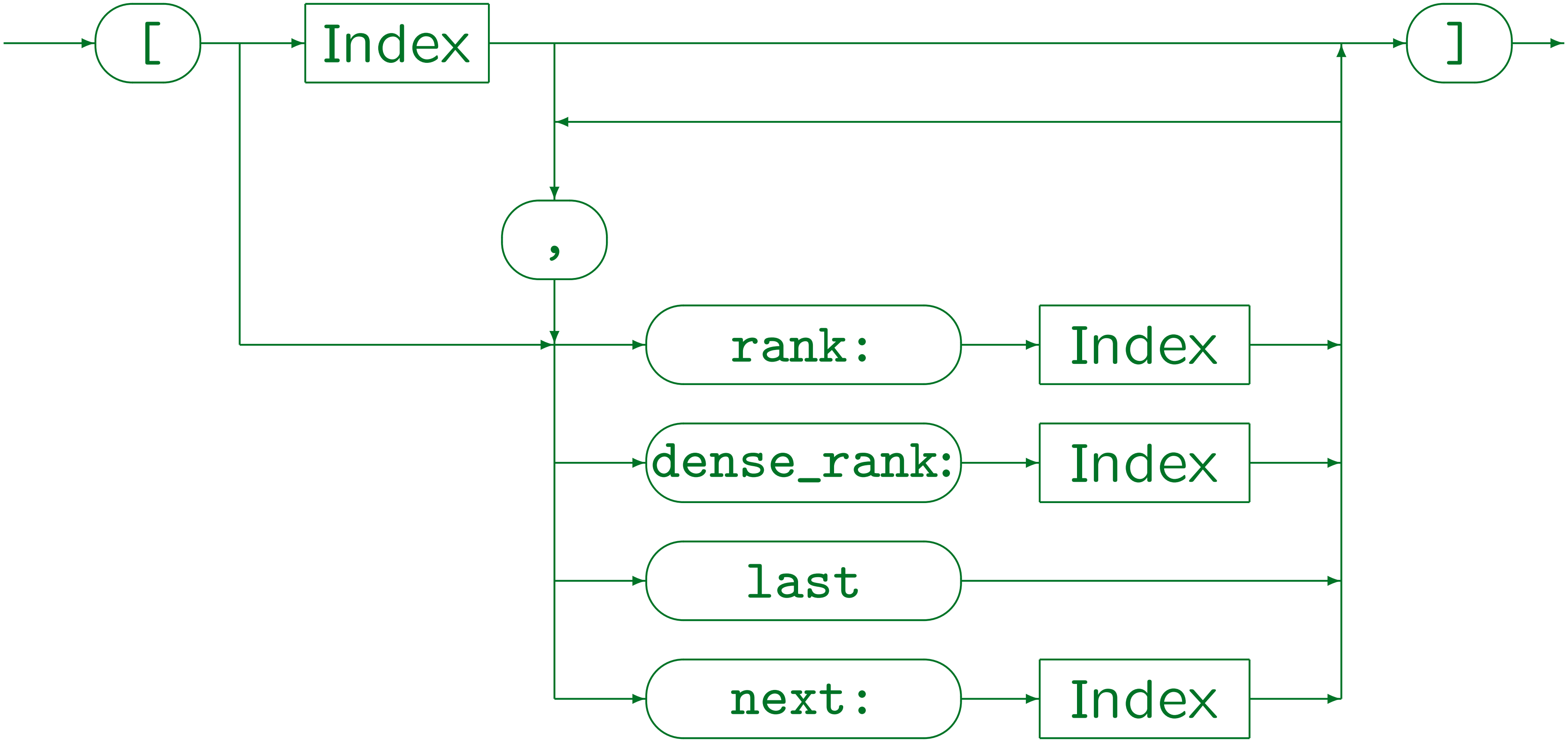
Syntax of Order Specification

Order-Specification:



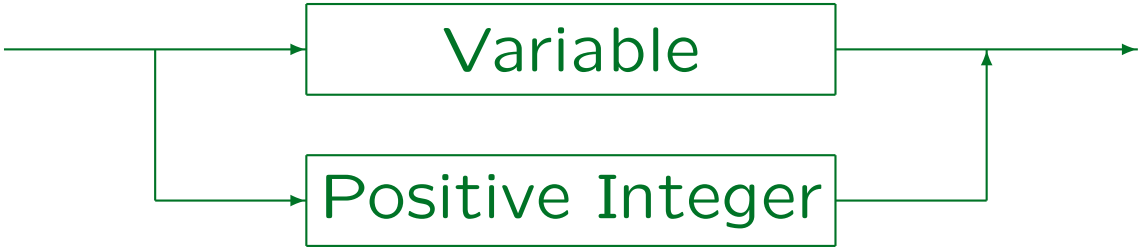
Syntax of Index (1)

Order-Position:



Syntax of Index (2)

Index:



Variable

Positive Integer

Simulation of Negation

- Negation can be simulated, e.g. by adding a dummy element which will certainly be last and checking whether it is also first.
- E.g. the rule

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

  can be expressed with order predicates as

  ```
  not_r<X|2>(yes, X) ← q(X).
  not_r<X|1>(no,  X) ← r(X).
  p(X) ← p_aux[1](yes, X).
  ```

Order-Spec in the Body?

- Helps to avoid auxillary predicates:

```
answer(EName, Sal) ←
    emp<^Sal>[N](EName, Sal, Job) ∧ N ≤ 3.
```

- However, consider this case:

```
answer(EName, Sal) ←
    emp<^Sal>[N](EName, Sal, Job) ∧
    N ≤ 3 ∧ programmer(Job).
programmer('Programmer').
```

The question is whether only programmers are considered when assigning row numbers, or row numbers are assigned first, and then programmers are selected. The problem is that it is no longer sufficient to consider a single assignment of values to variables when the rule is applied (aggregation is done here in the body, ∼ `findall`).

Declarative Output

Aren't there good solutions?

- A typical declarative solution is to use an IO state argument (accumulator pair):

```
main(IOState_in, IOState_out) :-
    io.write_string("Hello, ", IOState_in,
                               IOState_1),
    io.write_string("World!",  IOState_1,
                               IOState_2),
    io.nl(IOState_2, IOState_out).
```

This is from the Mecury Tutorial. Mecury has a special syntax that simplyfies this a bit, and also checks for determinismn (one cannot backtrack over IO).

Efficient Evaluation (1)

- A tuple stream (cursor/iterator) interface for predicates/relations is used.

  As usual, we try not to materialize intermediate relations in order to save memory (unless the same relation is used several times and it is cheaper to store it than to recompute it, or we need explicit sorting).

- A tuple stream can be ordered by given arguments.
- I.e. the task is to translate the given rules for a predicate into e.g. C++-code for iterators that produce the tuples in the required order.

  In the end, one wants the `output`-tuples ordered by first argument, but it helps if the predicates used in the `output`-rules can be fetched in a specific order.

Efficient Evaluation (2)

- For several rules about one predicate, it is often obvious that all tuples produced by one rule come before all tuples produced by another rule.

```
homeworks_table<1>("<table>\n").
homeworks_table<2>("<tr><th>Points</th>...").
homeworks_table<3,X>(Y) ← homeworks_row[X](Y).
homeworks_table<4>("</table>").
```

- So we choose the rule evaluation sequence correspondingly (avoids work at runtime).

- Otherwise ordered tuple streams for the single rules can be efficiently merged.

Efficient Evaluation (3)

- Also the nested loop join preserves certain orders.

  The full version of the paper contains an unconventional variant that does the sorting in groups, and helps to preserve more order.

- Base tables might have indexes or be stored in sorted order (e.g., as a b-tree), then tuples can be accessed in certain sorted orders.

- Sometimes, explicit sorting is unavoidable, and it is an optimization task to find the optimal place(s).

  It does not necessarily have to be at the very end. Sorting also helps with merge joins.