Optimizations to Earley Deduction for DATALOG Programs[1]

Harry H. Porter III
Portland State University

October 16, 1985

Author's e-mail: harry@cs.pdx.edu
Author's Web Page: www.cs.pdx.edu/~harry

This paper is online at:
www.cs.pdx.edu/~harry/earley/datalog.pdf
www.cs.pdx.edu/~harry/earley/datalog.htm

Introduction

Logic programs containing no functors form an important subset of logic programs and are called DATALOG programs. This paper explores optimizations that speed the execution of the Earley Deduction algorithm [Pereira and Warren 1983, Porter 1986] for this special case.

Two DATALOG clauses that differ only in (1) the values of constant arguments and (2) the names of the variables are said to have the same schema. For example, these two clauses have the same schema:

```
p(X,a,b) :- q(Y,c,X).

p(U,d,d) :- q(V,e,U).
```

The optimizations described below are designed to improve performance of the algorithm when there are large numbers of clauses with identical schemata. While it may often be the case that a DATALOG program has large numbers of unit clauses with the same schema, the real saving arises because many clauses with the same schema are derived during a typical deduction. The first optimization is a data structure that saves space when this condition is met. We then look at ways to do the subsumption checking and the reduction and instantiation steps quickly using this data representation.

---

[1] Although written long ago, when I was a Ph.D. student at Oregon Graduate Center / Oregon Graduate Institute / OHSU, this paper was reformatted in May, 2009 and posted to the web at this time. Other than repairing typos, no substantive changes have been made.

Definitions

We first define the *key* of a clause, which is a string constructed from the names of the predicates appearing in the clause and from their respective arities. For example, the clause:

```
p(a, X, Y) :- q(Y, b), r(X).
```

has the key:

p-3-q-2-r-1

The hyphen (-) is a special character not appearing in predicate names and is used to avoid ambiguity since predicate names may contain numerals.

The *variable format vector* for a clause is defined to be a string containing information about which argument positions are filled by constants and about variable usage in the clause. For the clause

```
p(a, X, Y) :- q(Y, b), r(X).
```

we first extract the argument information as **a-X-Y-Y-b-X** and then compute the format vector as:

#-1-2-2-#-1

The character "**#**" appears in positions containing constants, regardless of what constant is actually present, and the variable names have been *normalized* by renaming them with numerals. The variables are numbered sequentially from 1 in order of first appearance in the clause yielding the substitution:

```
{ X←1, Y←2 }
```

Two clauses have the same *schema* if and only if they have the same key and the same variable format vector. The schema is formed by combining the key and format vector. The schema for this clause is written as follows:

p-3-q-2-r-1/#-1-2-2-#-1

Clause Representation

The set of derived clauses constructed during a typical Earley Deduction sequence will usually be very large and must be represented and accessed efficiently if the procedure is to be practical. To achieve this, the clauses will first be indexed by the clause key and, within a key, will be indexed on the format vector. Given the key and the format vector of a clause, the only

remaining information needed to fully specify the clause is a tuple giving the values of the constant arguments. When there are large numbers of clauses with identical schemata this representation saves a considerable amount of space.

For example, the following set of clauses

```
p(a, X, Y) :- q(Y, b), r(X).
p(c, X, Y) :- q(Y, d), r(X).
p(c, U, V) :- q(V, c), r(U).
p(a, X, Y) :- q(a, b), r(Z).
p(e, X, Y) :- q(e, e), r(Z).
p(a, X) :- r(Y, a), s(Y), t(X, a).
p(b, X) :- r(Y, b), s(Y), t(X, a).
p(b, X) :- r(Y, a), s(Y), t(X, b).
```

has the following indexed representation:

```
p-3-q-2-r-1
   #-1-2-2-#-1
      a b
      c d
      c c
   #-1-2-#-#-3
      a a b
      e e e
p-2-r-2-s-1-t-2
   #-1-2-#-2-1-#
      a a a
      b b a
      b a b
```

Equality Checking

When a new clause is created with the reduction or instantiation rules, the algorithm must check to see if it is subsumed by an existing derived clause before it can be added. This prevents the storage of redundant clauses. If this test is relaxed to an equality check (i.e., the new clause is not added if it is already present modulo variable normalization), the algorithm is still guaranteed to terminate. However, it may do more work since some additional redundant clauses may be added to the derived set that would have been thrown out by the full subsumption check.

For example, these two clauses are considered to be equal:

```
p(a,X,Y) :- q(Y,b),r(X).
p(a,V,W) :- q(W,b),r(V).
```

These two clauses are not equal. The second clause subsumes the first.

```
p(a,X,c) :- q(c,b),r(X).
p(a,X,Y) :- q(Y,b),r(X).
```

Using the indexed storage scheme, the equality check is fast. To do it, first look in the key index and then look in the format vector index. To avoid sequentially searching the tuple sets, these are also indexed making a quick inclusion test possible. Thus, the clause equality search can be done with 3 hash table lookups—essentially in constant time.

Whether or not relaxing the subsumption check to equality will be a gain depends on the individual DATALOG program. The check will be very quick but in general more clauses are added to the derived set and the computation associated with them may overwhelm the time saved with the fast equality test. In experimental trials, equality checking is faster than the full subsumption check but not by very much.

Subsumption Checking

In doing the full subsumption check, we first look first at the key and format vector for a set of clause tuples and pre-compile the subsumption check. This short sequence of compiled instructions can then be "executed" for each of the tuples very quickly.

The problem can be stated as follows: Given a *candidate clause*, is it subsumed by any of the *target clauses* in the clause database? To determine this, first compute the key of the candidate clause and examine one-by-one all format vectors in the clause database under that key. For each format vector, compile a sequence of instructions. The compilation may fail, in which case none of the clauses with that schema can possibly subsume the candidate clause. If the compilation succeeds, execute the instruction sequence once for every tuple listed under that format vector. The execution will either succeed or fail for each tuple. If we find a tuple for which it succeeds, it represents a target clause which subsumes the candidate clause and the subsumption check is complete. Otherwise, after checking all tuples, move on to the next format vector and repeat the compile-and-search step.

Next, we describe the compilation step by example. Consider the candidate clause, which happens to be a unit clause.

```
p(X, b, c, d, e, f).
```

Let the target clauses to be tested (which have the same key, **`p-6`**) all have the format vector:

`#-1-2-2-3-3`

Examples of target clauses matching the format vector are:

```
p(a, X, Y, Y, Z, Z).
p(b, X, Y, Y, Z, Z).
p(c, X, Y, Y, Z, Z).
p(d, U, V, V, W, W).
p(e, Z, X, X, Y, Y).
```

which are represented by the tuples:

```
a
b
c
d
e
```

Do any of the target clauses subsume the candidate clause? For simplicity this example compares unit clauses but the generalization to non-unit clauses is obvious.

The compilation proceeds from left to right in the candidate clause and the target format vector. There are a number of different cases, depending on what occurs in corresponding positions in the candidate clause and the format vector. Here, the first argument of the candidate is a variable (**X**) and the format vector indicates that the target clauses will contain a constant in that position. Thus, regardless of what the actual constant is, none of the target clauses can subsume this clause. Thus, the compilation fails immediately so we continue the search by moving on to the next format vector.

For a more interesting example, let's test to see if the following candidate clause:

```
p(a, b, c, c, X, X)
```

is subsumed by clauses with the format vector shown above and repeated here:

#-1-2-2-3-3

The first argument of the candidate is the constant **a** and the first argument of the target is a constant. Thus, when we have the tuple in hand, we need to make sure that its first argument is the constant **a**. If so, we can proceed with any additional tests for this tuple. If not, then this tuple does not subsume the candidate and we can move directly on to the next tuple. The sequence of instructions being built is represented as a bracketed list. After examining the first argument position, the sequence consists of the single instruction to perform this test:

[#$_1$ = a]

The "**#**" character has been subscripted with 1 to indicate that it is the first constant in the target tuple that is being checked. The only kind of instruction for a subsumption check, as it turns out, compares a value from the tuple to a constant from the candidate. The instructions in a sequence will be separated by semicolons so, for example, the result of some compilation might be:

[#$_1$ = a; #$_2$ = c; #$_5$ = c]

The second position of the target clause contains a variable which doesn't occur anywhere else in the target clause. Since it will match anything, it is unnecessary to consider what appears at that position in the candidate clause. It could be a variable or a constant; in either case, no instructions are generated.

The third argument of the target is an example of a variable which appears more than once in the target. The first occurrence is handled differently than the remaining occurrences. When a variable is first encountered, it doesn't have a value so we assign to the variable (variable **2** in this case) whatever appears in the candidate's third argument. In this case, it is the constant **c**. The other possibility—a variable appearing in the candidate—will be discussed shortly. To perform the variable substitution, substitute the constant **c** directly into the format vector replacing all occurrences of **2** to give **#-1-c-c-3-3**.

In the fourth position, a secondary occurrence of variable **2** is encountered. Since we did the substitution, we actually encounter its value, the constant **c** directly. We must check the candidate clause's corresponding position. If it is the same constant (as happens to be the case), then we are okay. If it is a different constant or a variable, then the compilation fails and we move on to the next format vector.

In the fifth position, we encounter the first occurrence of variable **3** in the target. This time the candidate clause contains a variable, **X**. Just as for a constant, we perform the substitution, giving the target format vector **#-1-c-c-X-X**.

The last position is another example of a secondary occurrence of a variable and is treated similarly to the fourth position. If the candidate clause contains the same variable, **X**, (in this case it does) we are okay. If the candidate clause contains a different variable or a constant then the compilation fails.

After completing the subsumption check, we are left with only one instruction:

[**#$_1$** = a]

We can now look through the tuples for a tuple satisfying this equality. If one is found, then we have found a clause that subsumes the candidate clause, otherwise we can move on to the next format vector within this key and repeat the compile-and-search step.

Since we assume there are few schemata relative to the number of tuples, since the equality tests are fast and since we will only need to examine the tuples associated with a few format vectors (i.e., those for the same key that don't fail during the compilation step) this procedure reduces the cost of the subsumption check substantially.

The Reduction Step

The next question is: Can a similar kind of compilation speed the reduction step? There are two cases to consider:

(i) Given a unit clause, reduce as many non-unit clauses as possible yielding another collection of clauses.

(ii) Given a non-unit clause, reduce it by as many unit clauses as possible yielding a collection of new clauses.

The problem is to generalize the subsumption check described above to unification: instead of comparing complete clauses, we are unifying literals.

Consider case *(i)* first. The head literal of the given unit clause (the *candidate* clause) is called the *candidate literal* and is to be unified with the selected literal of each of the non-unit (*target*) clauses. The selected literals are called the *target literals*. Furthermore, for each that unifies with the candidate, a new tuple representing the reduced clause must be constructed and added to the clause database under the appropriate indices.

Again, the algorithm is described by example. The candidate clause is:

```
q(a, b, b, U, U, V, V)
```

It must be checked against all clauses that have a seven-placed predicate named **q** as the first literal of the body. To find such clauses, another index (called the *selected-literal-index*) is maintained. First, use the key for the candidate literal (**q-7**) to retrieve from the selected-literal-index all those keys of the form *x*–*x*–**q-7**–*x*–*x*–.... One such key is:

p-3-q-7-r-3

and it will be used for this example.

Associated with this key are several format vectors. We will look at:

1-2-#-#-2-2-#-#-3-4-4-#-2

For example, the clause

```
p(W, X, a) :- q(a, X, X, c, c, Y, Z), r(Z, d, X).
```

represented by the tuple

```
a a c c d
```

can be reduced using this candidate.

Begin by numbering the constants in the target's format vector. This will help keep track of which one is which.

1-2-#$_1$-#$_2$-2-2-#$_3$-#$_4$-3-4-4-#$_5$-2

Next, look at the format vector for the candidate literal. The variables in the candidate are renumbered to avoid any conflicts with variables in the target:

a-b-b-5-5-6-6

These two format vectors are called the *target* and *candidate*, respectively. The compilation will try to unify the 4th through the 10th positions of the target with the corresponding positions in the candidate.

Begin with the first position of the candidate. It is the constant **a** and it is matched against **#$_2$** in the fourth position of the target. This generates the instruction:

[#$_2$ = a]

The second position of the candidate is also a constant (**b**) but the corresponding position in the target is the variable **2**. So we make the substitution on the full target format vector giving:

1-b-#$_1$-#$_2$-b-b-#$_3$-#$_4$-3-4-4-#$_5$-b

In the next position, a constant (**b**) appears in the candidate and a constant (also **b**) appears in the target as a result of the substitution. These can be checked for equality at compile time so no instructions are generated.

The fourth position of the candidate contains the variable **5** and the corresponding position of the target contains the constant **#$_3$**. This time the substitution is performed on the candidate's format vector to yield:

a-b-b-#$_3$-#$_3$-6-6

The next position of the target contains the constant **#$_4$** and the corresponding position of the candidate contains the constant **#$_3$** that resulted from the substitution made in the previous step. A new equality comparison instruction is generated:

[#$_2$ = a; #$_3$ = #$_4$]

comparing the 3rd and 4th position in the target tuple. Because the same variable (**U**) appears twice in the candidate literal, the corresponding two constants in every target clause must be equal for the unification to succeed. This test reflects this requirement.

In the sixth position, a variable occurs in both the candidate (**6**) and the target (**3**). Again, we perform the substitution. It doesn't matter whether the variable **6** is replaced with **3** or the **3** is replaced with **6**. The result of replacing variable **6** by **3** is the following candidate:

a-b-b-#$_3$-#$_3$-3-3

In the last position, we have two variables, **3** and **4**. The substitution can be done either way so we arbitrarily choose to substitute the **4** in place of the **3**. Notice that, because of the previous substitution, variable **3** appears in both the candidate and the target format vectors. Because this can happen with any variable, every substitution must scan both format vectors. The unification is now complete and the resulting target format vector is:

1-b-#$_1$-#$_2$-b-b-#$_3$-#$_4$-3-3-3-#$_5$-b

The next step is to create the reduced clauses. First, remove the selected literal from the format vector of the target clause to give:

1-b-#$_1$-3-#$_5$-b

From this, we see that the new clauses will have 4 constants. Rename the variables and the constant positions to get:

1-#$_1$-#$_2$-2-#$_3$-#$_4$

The following *tuple creation mapping* tells how the new constant positions are to be filled in:

#$_1$ ← b
#$_2$ ← #$_1$
#$_3$ ← #$_5$
#$_4$ ← b

Armed with the compiled equality tests and the creation mapping, we are now ready to examine each individual target tuple. For each, first perform the equality tests. If they are satisfied, add a tuple representing the reduced clause to the database under the schema:

p-3-r-3/1-#$_1$-#$_2$-2-#$_3$-#$_4$

The creation mapping tells what constant tuple to add, namely the one made from a **b**, the constant from the first position of the target tuple, the constant from the fifth position of the

target tuple and another **b**. [Of course, we must first insure that the new clause is not subsumed by existing clauses before it is added to the derived set.]

These operations—comparing and manipulating tuple values—are familiar from relational algebra. In fact, the process of executing the comparisons and creating new tuples representing reduced clauses for any tuples found to satisfy the comparisons can always be expressed using standard relational operators. If we label the positions of the target tuples with the attribute names A_1, A_2, ... A_5 and call the set of target tuples the relation r, the set of tuples representing the reduced clauses can be represented in the notation of [Maier 1983] as:

$$\delta_{A2,A3 \leftarrow A1,A5} (\pi_{\{A1,A5\}}(\sigma_{A2=a} \ (r[A_3=A_4]r))) \bowtie <b{:}A_1\ b{:}A_4>$$

Case *(ii)*—where a single non-unit candidate clause is reduced by a number of unit clauses—is very similar to case *(i)*. The unification is pre-compiled to give a sequence of equality tests. The schema of the new clauses and the tuple creation mapping are made from the candidate clause's schema instead of the target's schema. The target tuples are then processed just as before.

Instantiation

When, in the course of deriving clause, a non-unit clause is produced there may be an opportunity to use its selected literal to instantiate a program clause. To speed location of potentially matching program clauses, we maintain a *program-rule-head-index*. Given a non-unit candidate clause, we use the key for its selected literal as an index to determine the keys for all non-unit program clauses with matching keys for their head literals.

Once we have these keys, a similar compilation-execution technique will speed up the instantiation process. The derived clause serves as the candidate clause and the program clauses are the target clauses. For each format we compile the unification check just as we did for the reduction step and, for every target clause satisfying the checks, we create a new tuple representing an instantiation of the target clause.

Delayed Subsumption Checking

Another speed optimization is to delay the subsumption checking until the tuples are referenced. Consider using the clause

```
q(X, X).
```

to reduce target clauses with the schema **p-2-q-2-r-2/#-#-#-#-#-#**, for example:

```
p(a, b) :- q(c, c), r(d, e).
```

Each execution of the resulting instruction list will (potentially) create a new clause (such as **`p(a, b) :- r(d, e)`**) to be added to the derived set. For each, we will repeat the compile-and-execute step to check to see if it is subsumed by existing derived clauses before it can be added.

Clearly, one such reduction step can produce lots of clauses with the same schema each needing the subsumption check. By doing them all at once, we can save a lot of redundant compile steps. To be more specific, we partition each tuple set into two groups—those that have already had the subsumption check and those that have not. Then, every time we reference a tuple set, we check to see if there are any tuples that have not yet had the subsumption check. If so, they are all done at once. This necessitates a change in the subsumption compilation algorithm as described above since we have a set of candidates instead of a single candidate, but this generalization is straightforward.

The difficulty with batching the subsumption checking comes in the bookkeeping associated with the deduction algorithm. Remember that every time a new clause is created (by reduction or instantiation) we must first see if it is subsumed and, if not, check to see if the clause can be used to reduced or instantiate any other clause, or can be reduced by any other clause. This is done by maintaining a list of clauses that need examining. Every time a new clause is created we check to see if it is subsumed and, if not, add it to this list.

The approach taken here when the subsumption checking is delayed is to always add it to the list. Then, when the subsumption check is performed, it is removed from the list if it turns out to be subsumed by existing derived clauses. We must be sure not to use a clause that is subsumed to generate new clauses or else we will quickly get into an infinite loop when these generate other clauses that generate other clauses... and so on. Another nuance is that indices into this list must be maintained since batched subsumption checking generally removes large numbers of clauses and we can not afford to blindly search it for every one.

Implementation Results

We have implemented the basic Earley Deduction and all the optimizations discussed above using Smalltalk on the Tektronix 4404 personal workstation. The basic DATALOG optimizations (indexing, tuples and compiling the subsumption checking, reduction and instantiation) increased the speed of the more general Earley Deduction algorithm dramatically, by more than 10 times for some small DATALOG programs. Furthermore, the improvement in speed increases as the total running time increases.

Replacing the subsumption check by the equality check also improved performance further (but only marginally) for all sample DATALOG programs run. Batching the subsumption checking improved (again only marginally) on the time taken when compared to the equality checking.

References

[Maier 1983]
Maier, David, *The Theory of Relational Databases*, Computer Science Press, 1983

[Pereira and Warren 1983]
Pereira F.C.N., Warren D.H.D., Parsing as deduction, ACL Conference Proceedings, 1983

[Porter 1986]
Porter, Harry H. III, Earley Deduction, Unpublished draft, 10 March 1986