Implementation Alternatives for Bottom-Up Evaluation

Stefan Brass
University of Halle, Germany

Deductive Databases (1)

- Deductive databases are integrated database/programming systems based on a declarative, logic programming style language (“Datalog”).
- SQL is a declarative, logic-based language, but supports only database queries (and updates etc.).
- Any serious database application needs also programming. Currently, SQL must be combined with a conventional programming language for this task.

  The need for programming has grown because of stored procedures, object-relational extensions and web applications.

Deductive Databases (2)

- So far, the performance of most prototypes is not very good, and this is an obstacle for a more widespread use in practice.
- Query/program execution in DDBs is done by
  - ◇ first transforming the given logic program so that only facts relevant to the query can be derived,
  - ◇ then (more or less) directly applying the T_P-operator in order to compute the entire minimal model of the transformed program with database techniques (bottom-up evaluation).

Our Approach

- I previously made a nice proposal for the transformation part (SLDMagic).
- Now I need a good bottom-up evaluation machine.
- My goal is to do this by translation from Datalog to C++.
- A student developed a prototype, but it turned out that there are several implementation alternatives (the prototype contains only the most basic one).
- I did performance measurements for these techniques with manual translations of several examples.

The Framework

- Every predicate/relation supports a cursor/iterator interface, i.e. there is a class p`_cursor` with methods
  - ◇ `void open()`: Open scan over relation.
  - ◇ `bool fetch()` Get first/next tuple (fact).
  - ◇ T `attr_`i`()`: Get value of i-th attribute/column.
  - ◇ `void close()`: Close the scan.

    Sometimes, also `push()` and `pop()` are needed to store the state of the cursor.
- If a relation has special access structures (e.g. B-tree index), there are additional cursor classes for specific binding patterns (then `open` has args).

Method 1: Materialization

- Program evaluation is done rule by rule.

  In a sequence that is computed by topologically sorting the "depends on" relation. Of course, recursive rules are applied iteratively, using deltas of relations (seminaive evaluation).

- Each rule is completely evaluated (on all matching facts for the body literals), before execution switches to the next rule.

  For recursive rules, only the facts that exist at that time can be used.

- Every derivable fact is stored until the end of program execution.

  Or at least until the last rule is applied that might use the fact.

Method 2: Pull

- Facts are computed only on demand (when `fetch` is called, this causes `fetch`-calls for body literals).
- Facts are not explicitly represented in memory.

  The `attr_`i-methods permit to access their attributes/arguments, but these values are typically spread in memory (in the fact or rule that introduced the value — no copying is done).
  Facts exist only until `fetch` advances to the next fact for that rule.
- Recomputation may be necessary.
- No recursion ($\rightarrow$ combination with other method).
- Standard method in databases to avoid storing large intermediate results.

Method 3: Push

- Derived facts are immediately used to derive more facts with other rules. Backtracking is needed if a fact can possibly be used in different places.
- Only a single fact of each derived predicate evaluated with this method is stored in memory.

  Unless duplicate elimination is needed (e.g. for cyclic recursions).
- This method works only with rules containing only one body literal with a predicate defined by rules.

  But: (1) SLDMagic produces such rules. (2) A combination with other methods is possible, e.g. materialization can be used for some predicates (which then no longer count for the limitation).

Experimental Results (Test 1)

```
p3(X, Z)  ←  p2(X, Y)  ∧  q(Y, Z).
p2(X, Z)  ←  p1(X, Y)  ∧  q(Y, Z).
p1(X, Z)  ←  p0(X, Y)  ∧  q(Y, Z).
p0(X, Z)  ←  r(X, Y).
```

$$\mathtt{q} := \{(i, i+1) \mid 1 \leq i \leq 5000\}$$
$$\mathtt{r} := \{(i, i) \mid 1 \leq i \leq 5000\}$$

Method	PC	Server	Sun
1: Materialization	515 ms	380 ms	430 ms
2: Pull	937 ms	320 ms	930 ms
3: Push	562 ms	250 ms	950 ms
3b: Push (local)	515 ms	250 ms	490 ms

Experimental Results (Test 2)

```
p3(X, Z) ← q(X, Y)  ∧ p2(Y, Z).
p2(X, Z) ← q(X, Y)  ∧ p1(Y, Z).
p1(X, Z) ← q(X, Y)  ∧ p0(Y, Z).
p0(X, Z) ← r(X, Y).
```

$$\mathtt{q} := \{(i, i+1) \mid 1 \le i \le 5000\}$$
$$\mathtt{r} := \{(i, i) \mid 1 \le i \le 5000\}$$

Method	PC	Server	Sun
1: Materialization	781 ms	830 ms	1950 ms
2: Pull	>1h		
3b: Push (local)	531 ms	290 ms	340 ms

Conclusion

- There are several different ways how to implement bottom-up evaluation (more than the three above).
- The best method depends on the input program, as well as the hardware and the compiler.
- Different methods can be combined.
- Next goals: Implement the transformation for these three and possibly other methods, develop heuristics for choosing a method, experiment with different data structures for storing relations.
- `http://www.informatik.uni-halle.de/~brass/botup/`