Virtual Views and Partial Materialization on Demand

February 7, 1997

Abstract

Work in progress....

1 Goal

We would like to be able to handle virtual views. A view defines a new database. If the view is virtual, this database is not materialized. To handle virtual views, we could move in two directions:

- Develop query rewrite techniques that will replace references to view objects by references to the "real world" objects they are derived from. That way a query that refers to a view can be processed by the query processor without having to materialize its "context" (ie, the view).
- Develop view rewrite techniques that will allow for smarter on-demand materialization, by pushing query conditions in the view definition, thereby allowing us to materialize only a fragment of the view.

The second strategy is more feasible, for the following reasons:

- The semantics of the view definition language are "heavier", in particular the `With` statement, making rewrite rules complicated and inefficient[1] (in the best case).
- Query rewrite rules have to be equivalence preserving. View rewrite rules only have to result in a "contained" view, *i.e.*, a view that is a "subset" of the originally defined. This fragment will be materialized and on it the query will be executed. So, if V is the view definition, Q is the query and V' is rewritten view definition, we only need to prove that $Q(V) = Q(V')$. Our requirement is that $V' \subseteq V$.

On the other hand, the view-rewrite/partial materialization strategy has a major drawback: the problem of reusing the materialized fragments. This problem relates to the answering queries using views problem, in that we want to know whether, for example, all our fragments make up the whole view. More work is needed in that direction[2].

1.1 Side issues on view specification

A few small items to add to the view specification language document.

[1] using the `With` we can "define" very complicated subgraphs.
[2] blatant understatement.

- As we have seen, in the case of a self-referential view specification the order of execution of statements matters. Of course Lorel is a declarative language, but in our context it is probably useful to state that within one `Define_View` block, the semantics correspond to the execution of the statements in the given order (and it's not, for example, the fixpoint).
- Arguments of the `With` clause are included in the result if they exist in the original DB, and are ignored otherwise.
- According to the specification of the `define <name>` statement, "each path expression [in the `With`] must begin with, or *have the same prefix as*, [a path expression] appearing within the Select_Expression." That contains an error (probably a typo): I guess the correct phrasing is that "each path expression [in the `With`] must *have as prefix* [a path expression] appearing within the Select_Expression."

  Additionally, let us consider the following view definition query:

  ```
  define View XXX
     define A := select f, f.Dept
                 from   .... f
                 where  f.Dept.name = 'toy'
                 with f.Dept.floor
  ```

  Let us try to replace `f.Dept` in the `select` clause by adding `f.Dept g` in the `from` clause. Now, it is ambiguous whether our original example ought to mean still `with f.Dept.floor` or `with g.Floor`. I think (and Serge agrees) that it should mean the former, *i.e.*, if more than one path expressions in the select are prefixes of an expression in the `with` clause, then the shortest one is used to determine the semantics of the `with` clause expression[3].

2 View rewriting for fragment materialization

The source database can be found in Fig 1. Let us start with the following view definitions and queries:

Example 1.

```
DefineView SomEmps

define SE := select e
             from   People.Emp e, People.Emp e1, e.Dept d
             where  d.name = 'toy'
              and   e.name like '\%son'
             with   e.name, e.salary, e.dept.boss.name

end DefineView


select a.name
from   SomEmps!SE.emp a
where a.salary > 100,000
```

Example 2.

[3]of course, we can always write `define A := select f, g from ...f, f.Dept g where .... with g.Floor` if we want.

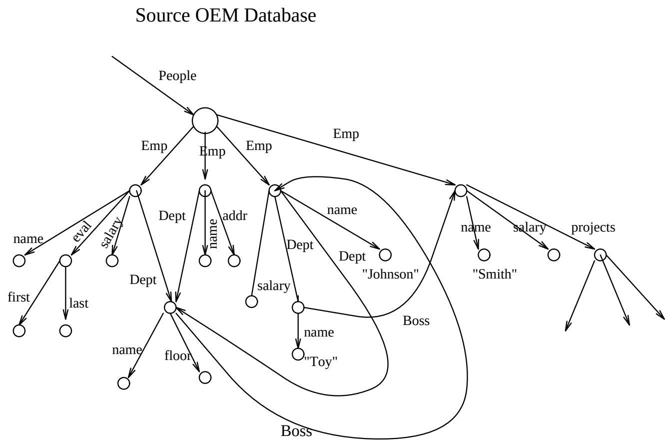

Figure 1: Sample Database

```
DefineView SoMorEmps

define SME := select  n,s
              from    People.emp e, e.name n, e.salary s
              where   e.age < 30
              with    n.first, n.last
end DefineView
```

```
select  c
from    SoMorEmps!SME.emp b, b.name c     --- correct only because of coercion
where   b.salary > 100,000
```

How do we go about rewriting the view? It is obvious how to find the name in the view definition that the query refers to. Now, let's observe that the query can refer only to objects appearing either in the select or the with clause of the define statement. Moreover, there are two cases to worry about:

- Say the query has an expression `SE.whatever a` in its from clause, like our first example. `a` points to the DB objects (that were included in the view) themselves (since the select clause of the define statement has one variable, `e`, in our first example). In that case, we can test whether the `SE.whatever` is valid by checking to see what is the edge pointing to `e` in the from clause of the query. If that edge is also `whatever`, then `a` in the query matches `e` in the view and we can replace it in any other expression it appears. So,in Example 1, the view definition can be rewritten as

  ```
  DefineView SomEmps

  define SE := select e
               from   People.Emp e, People.Emp e1, e.Dept d
               where  d.name = 'toy'
                and   e.name like '\%son'
                and   e.salary > 100,000
               with   e.name, e.salary
  ```

```
end DefineView
```

Paths not appearing anywhere in the query can safely be dropped from either the select clause or the with clause of the rewritten view definition. Moreover, we can probably eliminate from the query conditions that were "pushed" to the view – need to look into it a little more.

- Say the query has an expression `SME.whatever b` in its from clause, like our second example. `b` points to the constructed object that the select constructs (since the select contains a list of variables[4]) in our second example. In that case, we can perhaps "test" `b`, to see if it is "valid", by using the label coercion rules of Lorel. In the example, `b` points to the constructed object, and `emp` is the right label (if the query `said SME.? b`, we couldn't do any "testing"). In any case, `b.whatsnew` is "bound" to be one of the variables in the select clause of the view definition. Thus, `b.name c` binds to `n`, since it is ....`name n` in the from clause of the view definition.. It can be the case that more than one possible mappings of `c` exist.

Two more examples to illustrate that last point:

```
DefineView  XXX

define S := select n1,n2
            from   People.emp e, e.name n1, e.Dept.Boss.name n2
            where  n1.first = 'John'
            with n1.first, n1.last, n2.?
end DefineView
```

If the query asks

```
select n.first
from   S.? x, x.name n
where  n.last = 'Johnson'
```

we can rewrite the view as

```
define S := select n1,n2
            from   People.emp e, e.name n1, e.Dept.Boss.name n2
            where  n1.first = 'John'
             and   (n1.last = 'Johnson'
                    or n2.last = 'Johnson')
```

A couple more points: If more than one variable in the query matches to the same variable in the View definition, then we need to create new range variables in the view definition, that range over the same set as the existing variable.

2.1 Thoughts on more complicated cases

- if we have defined new edges, we CANNOT add the query constraints to the view, (and it doesn't sound very helpful to add them to the += statement), but we can add the += statement constraints (`where` clause) to the View name definition.
- if edges are deleted, we can still add the query contraints to the view definition as follows: [edge NOT IN (where clause of the delete statement) AND (where clause of the query)]

[4]We don't examine more complicated constructed results

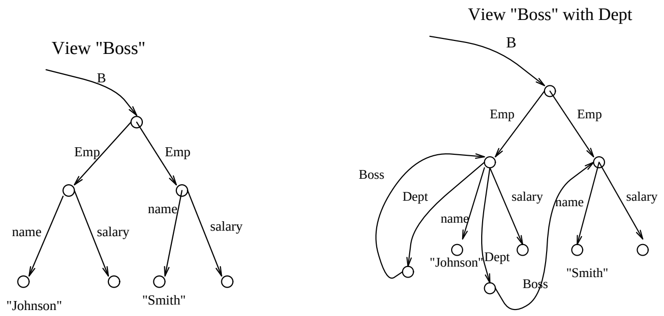

Figure 2: Examples of view definition

- if two view names are accessed by the query, and there are "joins" (ie, they both appear in one where clause term) then that expression can constrain the "second" name definition statement, but not the first.

3 What about query rewriting?

We will begin by trying to deal with the case where the view definition consists of `Define <Name>` statements only. More challenging issues towards the end of this. Let us consider the OEM database of Fig 1.

Also, let a view `Boss` be defined by the following statement:

```
Define_View Boss
   define B := select e2
               from People.Emp e1, People.Emp e2, e1.Dept d
               where d.Boss = e2
               with e2.name, e2.salary
End_Define
```

The view `Boss` appears in Fig 2. Next, we write a Lorel query refering to it.

```
select a.name
from   Boss!B.Emp a
where  a.salary > 100,000
```

That would be translated into a query on the source DB as follows (the source DB is assumed to be the default selectifier):

```
select e2.name
from People.Emp e1, People.Emp e2, e1.Dept d,
where d.Boss = e2
  and e2.salary > 100,000
```

or, to make it perhaps a little clearer,

```
select a.name
from People.Emp a
where a.salary > 100,000     *
  and a in (select e2
```

```
            from People.Emp e1, p.Emp e2, e1.Dept d
            where d.Boss = e2)
```

The above rewriting is correct because we can make sure in advance that the path expressions and/or labels involved in the view query are "ok", i.e. the view includes them, according to the specification (check that `a.name` and `a.salary` are ok, given the definition of `a` and the definition of the view).

Let us look at the following view:

```
Define_View Boss
   define B := select e2
               from People.Emp e1, People.Emp e2, e1.Dept d
               where d.Boss = e2
               with e2.name, e2.salary, e2.Dept.Boss
End_Define
```

The "contents" of the view are also displayed in Fig 2. The following Lorel query outputs the employees that are in charge of a Dept they belong to

```
select a.name
from   Boss!B.Emp a
where  a.Dept.Boss = a
```

[[[this uses simple path expression, no problem in and of itself]]] <==

The answer to that query is just

```
  answer
     name  'Jonhson'
```

The source Lorel query for this view query is

```
select e2.name
from   People.Emp e1, People.Emp e2, e1.Dept d
where  d.Boss = e2          <- this makes sure that e2 objs are in the view
  and  e2.Dept.Boss = e2    <- it's ok, because e2 is in the view,
                               e2.Dept is in the view and e2.Dept.Boss is in the view
```

If we write

```
select e.name
from   People.Emp e
where
  and  e.Dept.Boss = e
  and  e in  (select e'
              from   People Emp e1, e', e1.Dept d
              where  d.Boss = e')
```

that suggests a general way of rewriting the query. (we can push all nested queries in the where clause, if it makes any difference).

The general idea is to constrain every object at the end of an unknown edge, to be reachable thru the view definition. (like a "graph reachability" algorithm, that takes the where clause of the view into account. We know where to start from, since it's easy to check that used `names` are correct).

The problem with the above is that it's quite procedural.

There are more problems when features such as constructing results and generalized path expressions come into play. It seems the idea mentioned above can work with generalised path expressions, although it is going to be computationally expensive in the presence of Kleene closures (of course)

With constructing results, the problem is that they refer to new objects (and edges).

3.1 What about constructed results?

It is not clear how to handle virtual views with constructed results. For example, let's consider a query found in [AHV95]. Assume we have a CINEMA database, where each director has as children a name and number of movies, each movie a title (which is a string) and a set of actors. Directors can also act, so there is object sharing. The following view

```
Define_View CINEMA2
  define No_act = select Director: d.Name,
                         Movies: (select Title: m.Name,
                                         Actors: (select a.Name
                                                  from a in m.Actors)
                                  from m in d.Directs
                                  where d not in m.Actors)
                  from d in Director
```

explicitly *wants* to create a *new* complex value, that has the same structure as the CINEMA DB and only contains info about some films. It's counterintuitive to try to access source data to answer it. How to do it is also not obvious.

More name definitions: Having a view definition where more than one names are defined, and previous names are used in a `Define_View` block seems to present no additional problems. We need to use the dependency graph to do the "translation" in the right order.

Adding-deleting edges: For views containing `+=` and/or `-=` statements, the strategy outined earlier also seems promising, since they are only constructing edges. More work is needed.

4 Formalising the above