input:

1. labelled graph with values on some leaves
2. view query and view
3. update set: set of new vertices + added edges + removed edges + change of values [from DOEM paper]

output: the set of changes to the view.

Remark 0.1 Warning: this is a case analysis. It is hopefully complete so that the actual algorithm can be derived from this. We use a somewhat sloppy notation that is an extension of Lorel. This is to precise the semantics. A question is whether we can actually use Lorel queries to compute the maintenance queries. □

Remark 0.2 The change is decribed using atomic changes. It is a different (interesting) story if the changes are given by update statements. In that case, some compile time optimization can be achieved. □

1 White Zone

We treat the following views:

```
select X1,...,Xn
from <paths specifying X1,...,Xn,Y1,...,Yn>
where <conditions of the form Zi comp val or Zi comp Zj
       where Zi is Xi or Yi>
```

From the from clause we obtain an acyclic graph G with variables for vertices and an A-edge from Zi to Zj if the from clause contains Zi.A Zj.

```
select X
from ROOT.A Y, Y.B Z, Z.C X
where X > 14
```

insertions of B

```
WHITE += select X
         from ROOT.A Y, Y.<add>B Z, Z.C X
         where X > 14
```

If we don't have the DOEM but just a relation "NewBEdge", we can use it as follows:

```
WHITE += select X
         from NewBEdge(Y,Z), Y = ROOT.A, Z.C X
         where X > 14
```

Remark 1.1 The previous query was *not* correct. Note that it binds twice variable Y. A more correct one would be:

```
WHITE += select X
         from NewBEdge(Y,Z), Y1 = ROOT.A, Z.C X
         where Y1 = Y and X > 14
```

If we use queries for maintenance, the optimizer would have to be smart enough to (i) get Y, (ii) get bindings for Y1, check Y1 = ROOT.A using the lindex on Y1. □

Algorithm: start from the inserted B-edges and in the graph G follow E-edges with E-subfields and go backward F-edges in G using the lindex. This gives you the valuation to be tested for new results.

deletions of B

```
WHITE -= select X
         from ROOT.A Y, Y.<remove>B Z, Z.C X
         where ( select Y'
                 from ROOT.A Y', Y'.B Z'
                 where Z'.C = X and X > 14 ) = emptyset
```

Algorithm: start from the deleted B-edges and in the graph G follow E-edges with E-subfields and go backward F-edges in G using the lindex. This gives you the valuation to be tested for results to remove. Results should be reverified before being removed.

Value Updates This can come from insertions/deletions or change of atomic values. We know which ones have been modified.

Possible inserts:

```
WHITE += select X
         from ROOT.A Y, Y.B Z, Z.C<modified from OLD> X
         where X > 14 and OLD <= 14
```

Removals:

```
WHITE -= select X
         from ROOT.A Y, Y.B Z, Z.C<modified from OLD> X
         where X <= 14 and OLD > 14
```

But now for the query "select Y from ROOT..."

```
View RISKY =
     select Y
     from ROOT.A Y, Y.B Z, Z.C<modified from OLD> X
     where (X <= 14 and OLD > 14) OR
           (X > 14 and OLD <= 14)

WHITE := ( WHITE minus RISKY ) union
           select Y
           from RISKY Y, Y.B Z, Z.C X
           where ROOT.A = Y and X > 14
```

Let us see an example where a complex object is modified:

```
select X
from ROOT.A Y, Y.B Z, Z.C X
where X.D = 12
```

If we add a D edge to an object, this modifies it:

```
WHITE += select X
         from insertD(X,W), Z.C X', Y.B Z, ROOT.A Y
         where X = X' and W = 12
```

Again we get X, W and by lindex, Z and Y.
Algorithm: start from the objects being modified. Discard modifications that do not change anything (e.g., X= 24 to 35). See which vertices in the G graph could be affected by these modifications. Note that with a condition such as X.A = 5, an object is modified if A edges are added or removed.

Use the lindex and the objects to find the riskies. Then remove the riskies from white and reinsert the ok ones.

Optimization: Use of dataguide

Problem: techniques to use to reduce the size of the modified objects of interest by considering the paths leading to them. Interesting objects?

We need to have "some information" about the access to the modified objects.

The change of values could be $(oid, oldV, newV, att)$. We used the old and new value to see how they change the view. *att* is the attribute (could be a path) that lead to that object. E.g., H in:

```
update X.H := 5 ...
```

We now that there was an H edge to that object.

We could have a data structure that gives the pair of $(att1, att2)$ of labels that can point to the same objects (something like "compatible roles"). For instance if we find that D and H are not compatible, the previous update cannot change the above query result.

This is my understanding of interesting paths.

Optimization: Storing the valuation

If the set of valuations for a view is relatively small, it may be interesting to store it. Think of it as a table. Changes of values are more immediate, we need only to check the rows of the table that changed.

2 A problematic example

```
select X
from ROOT.B Y, Y.A Z, Z.C X
where X.A = 12
```

The problem here is that an addition of an A-edge modifies both the set of valuations and the set of values, so the where clause.

3 Black Zone

We maintain some auxiliary sets, one per variable in the with clause. This together with WHITE suffice to decide what edges to put in the view. The maintenance of these sets is similar to the problem solved by the WHITE zone (simpler - no condition).

```
select X
from X in WHITE
with X.A XA, X.H XH, X.F XF, XF.B XFB, XF.H XFH
```

We just have to compute the sets BlackA of vertexes that can be reached from WHITE with an A edge, and similary BlackH, BlackF, BlackFB, BlackFH. WHITE and these sets give us the population of the view. It is easy to obtain from them the edges.

For instance the expression for BlackFB

```
select Z
from X in WHITE, X.F.B Z
```

On insertion in WHITE: add

```
select Z
from X in New-WHITE, X.F.B Z
```

On removal of WHITE: remove

```
select Z
from X in New-WHITE, X.F.B Z
where NOT EXISTS X in WHITE : ( Z in X.F.B )
```

Addition of F-edges: add

```
select Z
from X in WHITE, X.<added>F.B Z
```

Removal of F-edges:

```
select Z
from X WHITE, X.<removed>F.B Y
and NOT EXISTS X in WHITE : ( Z in X.F.B )
```

4 Complex Paths

Consider some queries asking for objects with an Author edge:

```
select X            select X             select X
from ROOT.# X       from ROOT (.B)* X    from ROOT (.A.B.C)* X
where X.Author      where X.Author       where X.Author
```

Any addition of an edge may cause new objects to enter the view.

First one, on addition of B edges:

```
WHITE += select X
         from NewBEdge(Y,Z), ROOT.# Y, Z.# X
         where X.Author
```

We start from the new edge and follow links from there. (Observe that ROOT.# Y is not superfluous. We may get some unreachable objects in the instance.) Starting from X and going back would come down to reevaluating the query.

For the second, the from becomes:

```
         from NewBEdge(Y,Z), ROOT (.B)* Y, Z (.B)* X
```

So if we maintain a datastructure for (.B)*, it is two lookups in it.

For the third, on insertion of B edges the from becomes:

```
         from NewBEdge(Y,Z), ROOT (.A.B.C)* Y1, Y1.A Y,
         Z.C Z1, Z1 (A.B.C)* X
```

[Let us assume that we do not nest stars as in (.A(.B)*)*.]

More examples:

```
select X                      select X
from Name (."sec%") X         from ROOT .C (.A | .B) X
where X.Author                where X.Author
```

For the first, value updates have no effect. Edge deletions/additions matter only if the label matches "sec%" or Author.

For the second, edge deletions/additions of A, B, Author matter only. On insertion of B, the from becomes:

```
          from NewBEdge(Y,X), ROOT.C Y
```

Optimization

How can we speed up this computation with regular paths? Consider

```
select
from Name.A X, X <path P> Y, Y.C Z
where Z = 45
```

What do we need to do:

1. if an edge is removed (and the label matches a label of the path), we need to know the pairs (X,Y) such that there there was a path P from X to Y and there is no more, or similarly for addition and X, Y that become P-connected.

2. Given X, we need to know all the Y such that there is a path from X to Y;

3. Given Y, we need to know all the X such that there is a path from Y to X.

In general, this data structure may be quite large, we may decide to maintain it only if it is below a certain size.

Remark 4.1 If the path expression is more complicated (nesting of stars), there is a little (not much) complexity since given a regular expression R and a label (added/removed edge), we need to find R1,R2 s.t. R1 a R2 included in R. [Easy to do on FSA.] □