# Cascade delete

Joop Ringelberg 27-10-20 Version: 1

## Introduction

The user can remove roles and/or their bindings. This removes edges from the graph of contexts and roles. Intuitively we understand that a situation may arise where a context or role can no longer be 'reached' by the user[1].

But what does that mean, exactly? We will explore two intuitively appealing definitions and show only one of them is right. First, however, we discuss why it is important.

## Why do we even care?

Is it important? Such contexts and roles could be removed from cache and, indeed, from the database, without us ever missing them. But by the same reasoning, they don't seem to be in the way, either.

However, this is not true. Consider a group activity amongst friends, planned for some future date. John and Pete have both signed up. Now John withdraws from the activity. Then, sometime later, Pete does the same. Still later, their friend Mary adds John again. This will cause John to receive deltas that describe the (then) current state of affairs. Were the activity not removed from Johns database when he withdrew, his PDR *will apply them to the context as it finds it in the database*. As a consequence it would still list Pete as an active member, because John did not receive an update when Pete withdrew (John did not need to be informed because it was known at the time he did not participate). And the deltas sent by Mary's PDR merely describe the current state; not what has been removed over time. This clearly is unwanted and, even worse, the inaccurate information might spread to the others, too, under the right circumstances.

So we need to have the database contain only contexts and roles that reflect the current state of the world in an accurate way.

## A notion based on graphs

As we started with a discussion of reachability in terms of the graph of contexts and roles, it seems reasonable to define the concept in these terms. This might be a definition:

> **A context cannot be reached if none of its roles has a binding or fills another role. (I)**

[1] In this text, the term 'the user' refers to the System User role or any other role it fills, *for a given installation of the PDR*. This is the 'owning user', as distinguished from his peers (who, each in turn, obviously are 'the user' for their own installations).

And, indeed, this definition is sound; but it turns out to be insufficient. It has several issues:

1. Consider an isolated subnetwork of contexts without any user roles. Surely, the contexts and roles in that subnetwork should be removed. However, each of them may have bindings or fill other roles – in the subnetwork!
2. In a more technical vein, we represent role binding with edges leading both ways. However, it might well be that the user just has queries traversing the binding in only one way!
3. More conceptual: we represent just direct links between roles, but reachability is about *paths*. See the next paragraph *A notion based on perspectives* for an explanation.

## A notion based on perspectives

A user role can have a perspective on another role. When this role is calculated, the perspective traces a path through the graph of contexts and roles. In other words, calculated roles and properties make it possible for a user to look beyond the borders of his context. It is these paths we have to deal with when deleting roles and / or severing links between them.

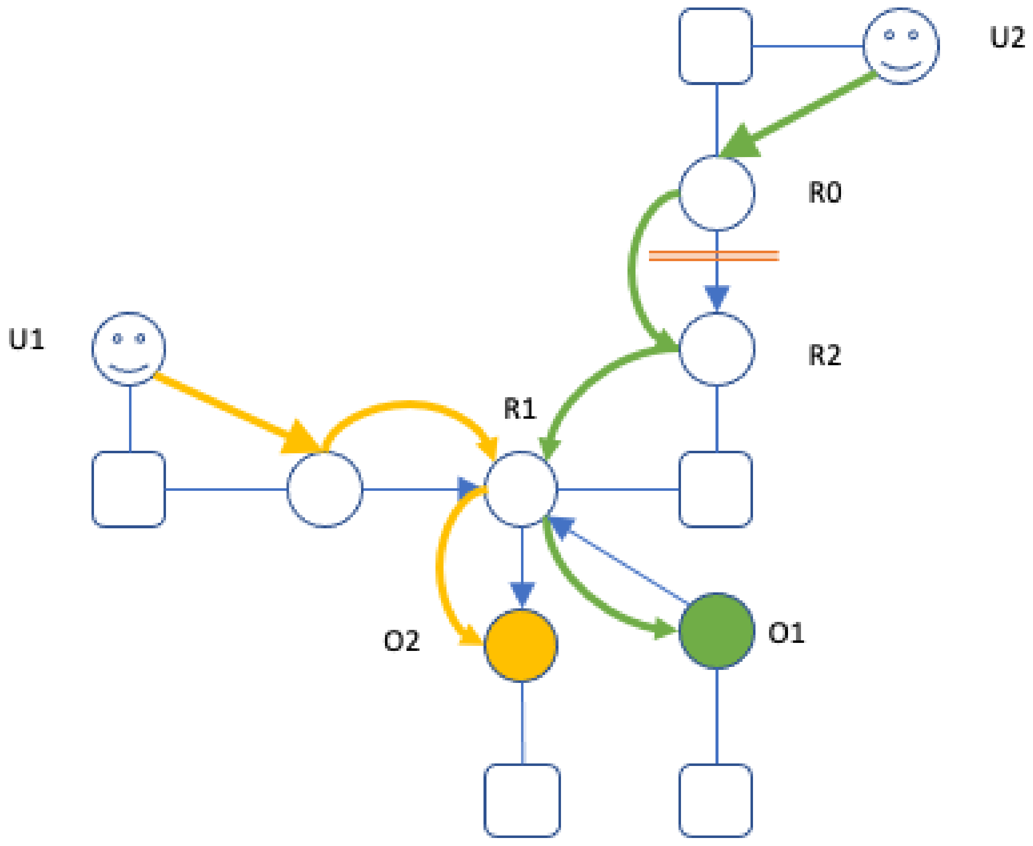

*Figure 1. Squares represent contexts; circles roles; smileys user roles. A pointed line represents a binding (bound node at the arrow). The yellow and green paths both visit node R1. When the graph is cut at the red line, U2 can no longer see O1. However, departing from O1, there is still a user node connected to it (U1). So the graph concept of connectedness is not the same as the perspective concept.*

As we see in Figure 1, users can 'see' nodes as traced out by query paths. Severing links on such a path may cause nodes to be lost out of sight. In this case, node O1 is no longer visible to any user if we cut the edge at the red line, and should be removed. Incidentally, node R2 should be removed as well!

However, we can still trace a path from O1 to some user (U1), suggesting we cannot remove O1. So the graph does not give us enough information to make the right decision. This is because we have to deal with paths, rather than connections.

This leads us to a second definition, both sound and complete:

> **A role cannot be reached if there is no role representing the user that has a perspective on it (II).**

Note that we use the type-level notion of perspective on the instance level, here. What we mean is that there is no (type level) query that leads from the user to the role (where each query starts, as it were, with a perspective on a role in the context – either enumerated or calculated).

## On computing cascade delete for roles

The example above illustrates what roles can be deleted once we severe a binding. Let's examine the situation again:

- The path consists of role instances R0, R2, R1 and O2 (in that order).
- We sever the binding between R0 and R2.
- As a consequence, R2 and O2 must be removed, while R1 must stay.

How do we decide what roles to remove?

The algorithm to follow consists of these steps:

1. Start with R2, the node that the binding arrow points to[2].
2. Now retrieve, from the type definition of R2, the *queries with a kink* that are stored for the `binder R0` step. A query with a kink has a backward- and forward facing part. The first step of the backward facing part determines the member of the role type where it is stored. Here, that first inverse step is `binder R0`.
   We find one such query in our example (drawn with green arrows).
3. Apply the sub-algorithm CAN THE ROLE BE REACHED? to R2 and that query.
   Since we've severed the link between R0 and R2, we will find that R2 can no longer be reached and should go.
4. Run the forward facing part of the query we've found to find roles that it visits. For each of these roles, run the sub-algorithm CAN THE ROLE BE REACHED? Here, these

[2] Actually, we will apply the algorithm also to the queries stored in R0 in the onRoleDelta_binding step, representing queries that run originally *from* R2 *to* R0. But in our example, there are no such queries!

roles are R1 and O2. We will find that R1 can still be reached (from U1), but that O2 can no longer be reached and should go.

#### Sub-algorithm: Can the role be reached?

Starts with a number of queries with a kink and a role instance.

1. Run the backward facing part of the queries we've found, starting with the role instance, to find the users sitting in the starting context, reaching out towards roles outside it.
2. *If none are found that represent the user, the role can safely be removed.*

## On computing cascade delete for contexts

This is actually really simple. If a context has no roles left, it can safely be removed! Thus we complement definition II with:

**A context cannot be reached if it has no roles (III).**

### The external role

In the current implementation, we treat the context and its external role as a unit when it comes to creating and deleting. However, the algorithm above applies to external roles just like any other role. So should we start deleting external roles?

We then may wind up in the situation where a context has no external role, but still has other roles (through which it is connected to the rest of the network). This may cause havoc in the implementation; it would be better to separate the tie between context and external role and only create an external role if it can be reached. That may be quite a consequential refactoring, though!

A pragmatic way of dealing with this issue is to *not remove an external role*. Instead, whenever we remove the last *other* role from a context, we check the external role. If it fills no other role, it is as good as gone; we can than safely remove both the context and the external role.

## Synchronization

Should we send deltas to peers for roles and contexts that we've deleted while cascading a seed change? No. Whether a role (and context) is still reachable is a matter of perspective. We only compute it for the 'owning user', hence we should send no deltas.

## Deleting roles

Above, we've discussed how to handle the removal of a binding. A similar algorithm should be followed when we delete a role instance from a context. Obviously, when we delete a role, we also remove its binding and the bindings of roles that fill it. The above algorithm applies to those cases.

However, for the role we remove, we should also check the queries with a kink stored in `onContextDelta_context` and `onContextDelta_role`, for inverted queries that respectively:

- have `context` as their first (inverted) step, or
- have `role R` as their first (inverted) step.

## Caveats

The discussion above exposes the general case. However, there are two exceptions to the rule.

### Database Query Roles

Database Query Roles (DBQ) are computed roles. They are semantically equal to context roles, but without a represented context role in their embedding context. When computed, they return a collection of external roles.

All of the above applies to those contexts and their roles. However, the administration of filled roles on their external role misses the context role that ties such a context to an embedding context. We cannot, therefore, decide that such a embedding context no refers to it – unless the end user explicitly removes such embedded context from it.

### Indexed roles and contexts

A context such as usr:MySystem or usr:MyChatApp can always be found, using its universal (indexed[3]) name. Because of that, we should never delete them in a cascade.

It is up to the modeler to let the user delete them by hand or not, or to have a bot do it for them.

[3] The term 'indexed' is somewhat confusing here. It was coined to mint the notion of entities that each end user refers to by the same name - not because there is a pointer to it from some other data structure!