Aspects require refinement of inverted queries

Joop Ringelberg 30-11-21, 28-01-22 Version: 2

General problem statement

The text “Query Inversion” introduces the concept of an Inverted Query. Briefly, this is a mechanism to ensure synchronisation. A perspective grants a user access to entities (roles, contexts, including bindings and property values). On the type level, we associate inverted queries with, say, a Property type that will help us find the User role instances that should be informed about a modification of the Properties’ value on a given role instance.

Aspects are nothing else but Context- and Role types that are *added* to other types, to enrich them with Role types (when adding a Context as Aspect) or Property types (when adding a Role as Aspect). We will call the property P of a role R an *Aspect Property* if the lexical context of P is not R, but a role AR of some other context AC. It helps to prefix the local property name with its lexical role context to recognise it as an Aspect Property: so AR$P is an Aspect property of R, while R$Q is a *local* property of R.

In this text I identify a problem that arises during synchronization using Inverted Queries for Aspect Properties and Aspect Roles.

The problem in detail

Consider Figure 1. It shows a somewhat contrived model for a family and the birthdays of father and mother, respectively (all contexts). Both birthday contexts use an Aspect Party that has a role Present with a property Price. The role Present Father uses Present as Aspect Role and so does Present Mother. The result is that both Birthday Mother and BirthDay Father have a Price property on their Present roles.

Furthermore, the Father role of Family is a calculated role in Birthday Mother and has a perspective on Present Mother. Notice the cross-over: Father has access to the Price property of the Present Mother role; Mother has access to the Price property of the Present Father role.

Consequently, this perspective is inverted and attached to the Price property, leading to Birthday Mother, naming (the Calculated) Father role as one that should be informed when Price changes. Symmetrically, we have an inverted query from the Price property to Birthday Father.

But remember that Father is calculated. So when we detect a Price change in Present Mother, we first find an instance of Birthday Mother and then calculate Father.

Now look at the red lines from Price to Father and Price to Mother. They consist of many hops from entity to entity and this reflects the query path that connects the property to the user role that should be informed when the property changes[1]. Obviously, when the price of mother's present changes (maybe a child sets it), father should know (but mother should not!).

We can now state the problem precisely. The *two* inverted queries are *both* stored with the definition of Property Price. When the actual property value of the actual role instance of Present Mother changes, we look up the inverted queries stored with Price and execute them to find the role instances that should be informed. So we find two queries. But, by coincidence, both queries will give a result when applied to either the price of Present Mother, or to the price of Present Father. Hence, we'll find that both Father and Mother will be informed when the price of either present changes – clearly an unwanted situation and not the one that was modelled.

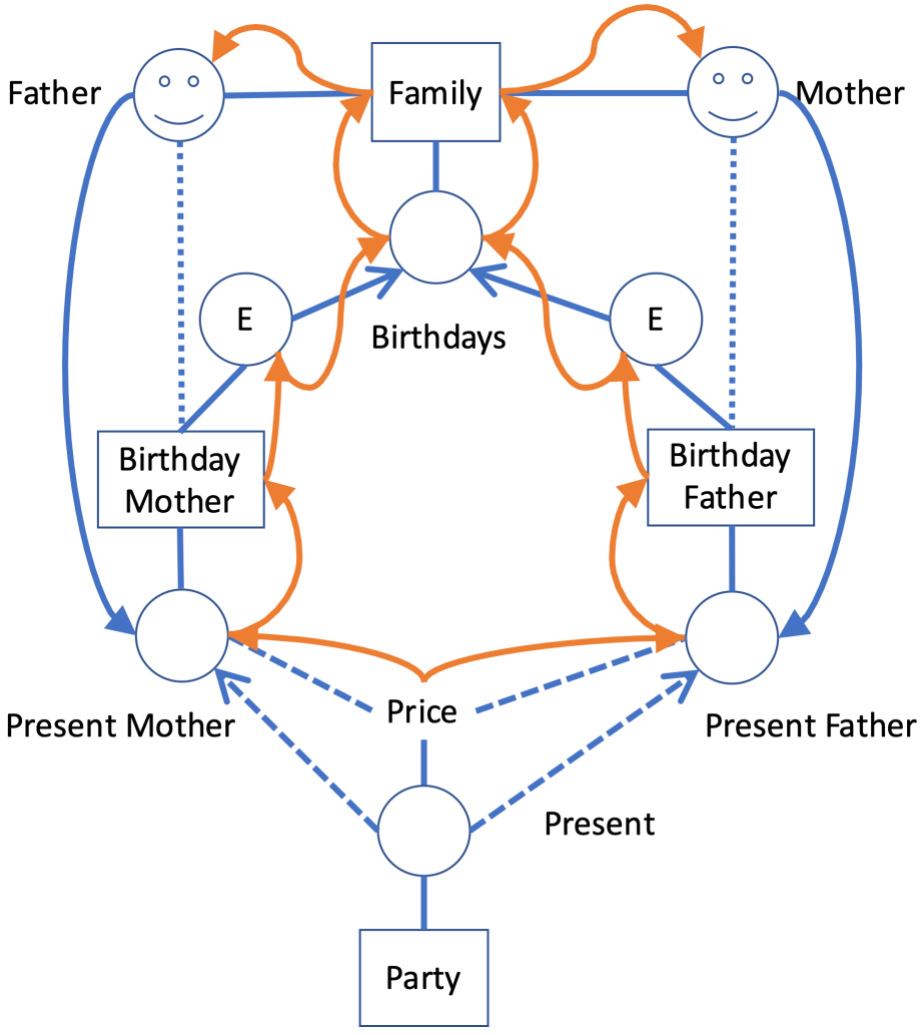

Figure 1. This model contains a synchronization problem for Price. The red lines are paths to follow when Price changes; the lines with long dashes represent an Aspect relation. The lines with short dashes represent the fact that the roles are calculated with respect to their contexts. So: Father is an Enumerated role of

[1] Consisting of two parts: the *inverted* path from price to Birthday Mother, and the *normal* path from that context to the Father role of Family.

Family, and a Calculated role of Birthday Mother, while Price is an Aspect Property of Present Mother (and Present Father, too).

Making sense of this situation

We should consider the property Price *as added to one role*, Present Mother, to be different from the property Price *as added to another* (Present Father). Think of them as templates, where their incorporation into other role types stamps them into something new. Price *as used in Present Father* is a different property from Price *as used in Present Mother*. Changing either of these different properties should not be mixed up with changing the other.

A similar thing holds for role Aspects. In fact, at the type level, we should identify roles with the combination of a context- and a role type: `RoleInContext`.

This is important when we analyze and use *paths through type space*. A query, and consequently an inverted query, is such a path. We should describe such paths in terms of Context types and `RoleInContext`s, rather than Context types and `EnumeratedRoleTypes`.

When a user makes a change to the structure of roles and contexts, she adds or removes a role to a context or fills (or clears) a role. We describe such a change in terms of a *Delta* that identifies either a context- and role instance, or two role instances. As a role instance representation contains a direct reference to a context instance, we can immediately derive a `RoleInContext` type combination from a role instance. Hence, we can identify with certainty two points of a path through type space. With those points in hand, we can find inverted queries that run through them and use those to find the peers that need to be informed.

We really should use two points rather than one. The Delta describes a path of length two; we want to make sure we only follow (inverted) queries that incorporate that entire segment. Were we to use just one of the two points, we would include all relevant queries, but would err on the other side by including paths we don't want to follow.

Solution

All in all we see that the current implementation (v0.12.0) is incorrect.

Separating groups of inverted queries conceptually

Inverted queries starting on a property

We should be able to distinguish between Inverted Queries on a Property type for the various EnumeratedRole types that they are added to. Currently, Inverted Queries are stored in an Array in the representation of Property. This we will change to an Object, where the keys represent `EnumeratedRoleType`s.

Inverted queries starting on a context

When a new role instance is added to a context, we must follow queries of which that segment from context to new role instance is a part. The relevant Inverted Queries are stored on the `Context` type in the `invertedQueries` collection[2]. A `ContextDelta` gives us both the context instance and the role instance. In principle, we need to identify the type of the role instance with a `RoleInContext`; but, unsurprisingly, its `Context` type part will by construction be the type of the Context instance. Hence, we can properly recognize `InvertedQueries` fitting our Delta by the `EnumeratedRoleType` alone, in the collection of inverted queries on the appropriate Context type.

A role type may have more than one instance in a context instance. Because we do only need to evaluate the new (or removed) segment, we do actually not apply the inverted query from the context instance, but from the new (or deleted) role instance. This is somewhat confusing. We treat these queries as if they start with the context step, while they actually do not.

Inverted queries starting on a role

Let's reconsider the various Inverted Queries that are stored with an `EnumeratedRoleType`. We categorize them according to their first step, that determines whether the path leads to the role's context, its filler, or the roles it fills.

For queries that start with a context step, the relevant queries in the model are those that start on the `RoleInContext` that we derive from the `ContextDelta`. But as we store the `InvertedQueries` on the `EnumeratedRoleType`, we can separate out the relevant inverted queries with the `Context` type alone, similar to the role step discussed above.

Inverted queries between two roles

The fills step and the filledBy step move between two role instances. We can derive a `RoleInContext` from both and that means we can find the relevant inverted queries by the *combination* of these two `RolInContext`s.

Representation of Queries

We represent queries with a QueryFunctionDescription that gives us the domain, range, function, and some meta-properties and the argument expressions that supply values to be bound to function parameters. Domain and range are constructed as an Abstract Data Type (ADT) of a base type and role domains are based on `EnumeratedRoleType`s. We see now that they must be based on `RoleInContext`.

[2] There is no need to make this name more discriminative, as only one step (the role step) leads away from contexts.

Runtime indexing

Let's illustrate the above with some examples.

Consider an Aspect Role Driver, to be added to both a Car and a Train context type. Suppose both contexts have other user roles that have a perspective on the Driver. This would establish inverted queries for both contexts on the Driver role. Clearly, we must be able to distinguish the queries for the Car context type from those for the Train context type. In other words, we have a `RoleInContext Train Driver` and a `RoleInContext Car Driver`.

Runtime indexing: context step

In runtime, how do we index? We should subdivide inverted queries that depart from a role instance with the `context` step according to the `RoleInContext` that is a combination of the `EnumeratedRoleType` and the `ContextType` that is reached. However, by construction, the role type will be the type that we store the queries on. So we can leave it out and just use the `ContextType` to subdivide the inverted queries.

That is, we can choose the right subcollection of the `contextInvertedQueries` on the Driver role by using either the Train or the Car `ContextType`.

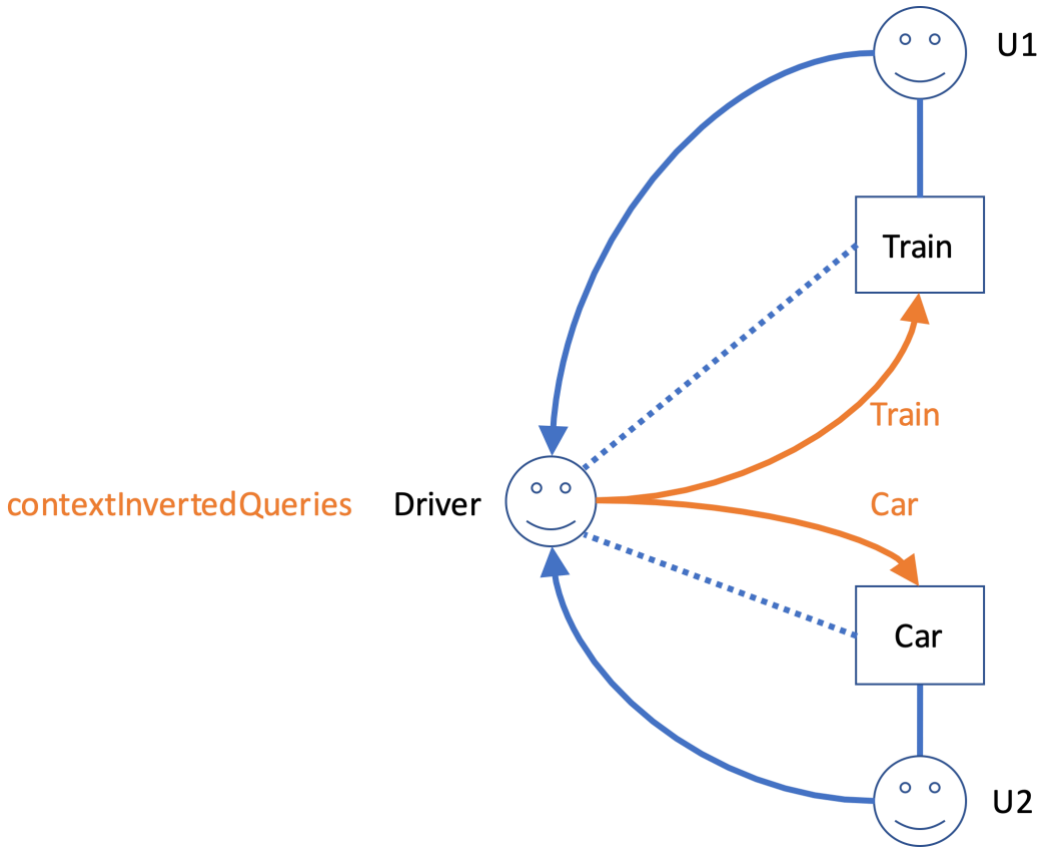

Figure 2. The inverted queries stored in contextInvertedQueries (orange lines) in the Driver Role type should be indexed by the context type of their endpoints, or, equivalently, the context type of the role type of departure.

Runtime indexing: role step

Inverted queries that, conceptually at least, start with the role step, are stored with a `Context` type. They are subdivided according to the `EnumeratedRoleType` that they lead to (as explained above).

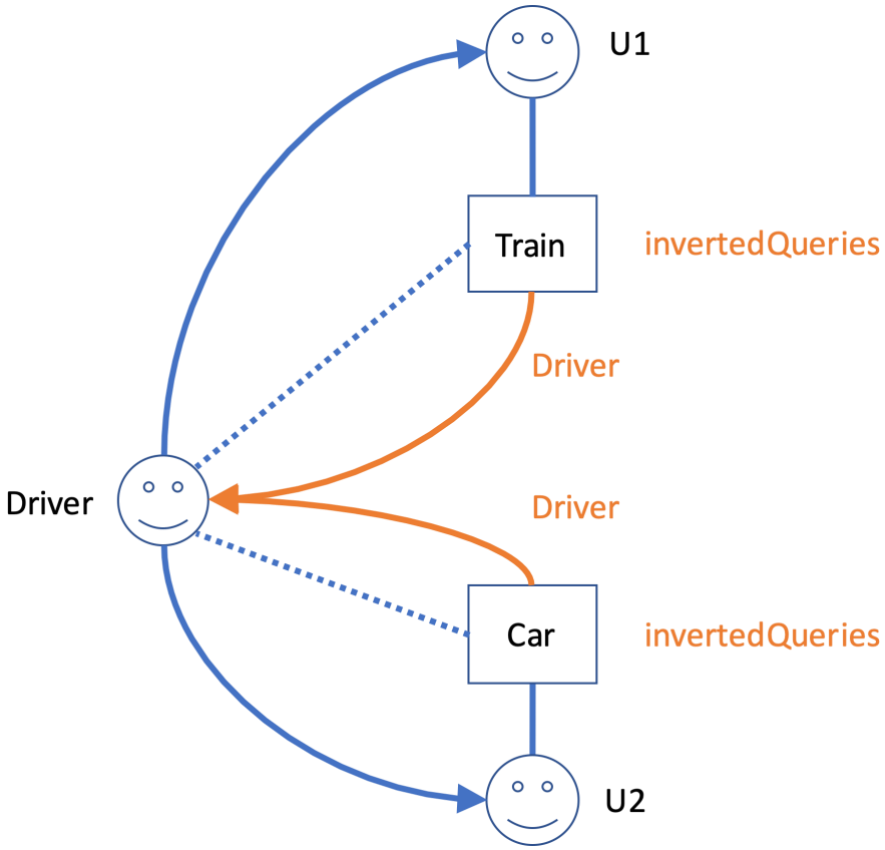

Figure 3 In this figure, the perspectives are the other way round (this is a contrived example, as Aspect roles would not have perspectives on roles in contexts to which they are added. However, it serves to illustrate the principle. We see that inverted queries, stored with the Context type, should be indexed with just the EnumeratedRoleType they lead to.

Runtime indexing: filler step

Inverted queries that depart from a role instance with the `filler` step[3], are stored in the `filledByInvertedQueries` collection on the `EnumeratedRole` type. What subcollections should we distinguish? Remember that we must make sure that we only apply inverted queries that contain the segment that is described by the Delta. This is now defined by *two* role instances, or, in other words, *two* `RoleInContext` instances.

As the step has a direction[4], we will combine (apply) the `RoleInContext` instances in that order. Notice that, by construction, the first of these will always have the type of the `EnumeratedRole` that the collection is stored in. So we could do with just its

[3] Previously: `binder` step.
[4] From Filled to Filler.

`ContextType`. This is not true for the second `RoleInContext` instance: either of its components may vary freely. Thus, we have three keys, applied in this order, to find the right subcollection of inverted queries:

1. The `ContextType` of the Filled role in the Delta, or, in type time, of the `Domain` of the `QueryFunctionDescription`;
2. The `ContextType` of the Filler role in the Delta, or, in type time, the `ContextType` of the `Range` of the `QueryFunctionDescription`;
3. The type of the Filler role in the Delta, or, in type time, the `EnumeratedRoleType` of the `Range` of the `QueryFunctionDescription`;

For the implementation we have a number of representation choices, ranging from three nested Objects to a single Object with a key that is the combination of the string representation of the three keys, to a Map with a key constructed of the three types.

Runtime indexing: fills step

Inverted queries that depart from a role instance with the `fills` step[5], are stored in the `fillsInvertedQueries` collection on the `EnumeratedRole` type.

The reasoning is symmetrical to that for the filler step. Again, we must use two `RoleInContext` indices; again, we can ignore the `EnumeratedRoleType` of the first of these. However, the direction is inverted. So we have:

1. The `ContextType` of the Filler role in the Delta, or, in type time, of the `Domain` of the `QueryFunctionDescription`;
2. The `ContextType` of the Filled role in the Delta, or, in type time, the `ContextType` of the `Range` of the `QueryFunctionDescription`;
3. The type of the Filled role in the Delta, or, in type time, the `EnumeratedRoleType` of the `Range` of the `QueryFunctionDescription`;

Again, like with the `role` step, the fills step has cardinality greater than one (it can fill many other roles). For this reason we shorten the inverted query and the runtime applies it to the filled role – not to the filler role.

[5] Previously: `binder` step.

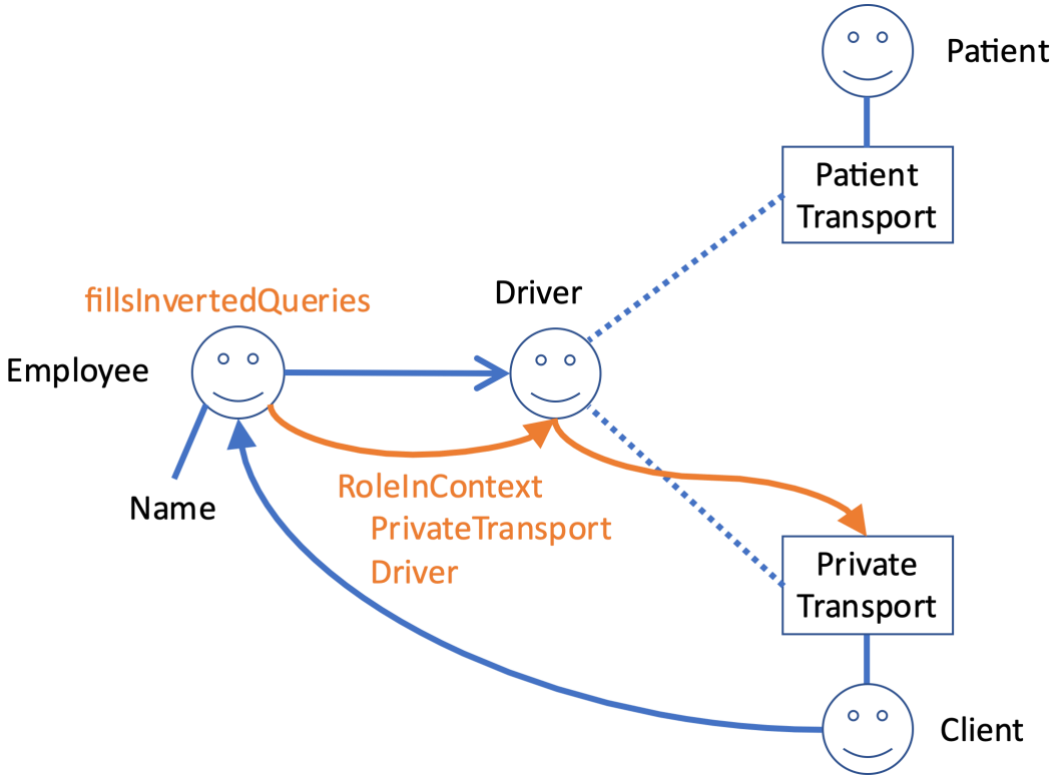

Figure 4. The Employee (of, say, a taxi company) fulfills the Driver role in both transport of severely ill people and for privatel rides. Now suppose (for the sake of the argument) that the Client of such a private ride has a perspective on the Driver, including his name, but the Patient does not. The inverted query that is stored in the fillsInvertedQueries collection of Employee actually starts on the Driver role instance (because the binder step has cardinality greater than 1). But clearly, we should index this collection of inverted queries with `RoleInContext PrivateTransport Driver`*, order to prevent the system from informing Patients when the Driver role is filled. Actually, we should also use the Context Type of Employee to cover the situation where Employee was used as Aspect as well.*

Compile time indexing

In compile time, we invert the *description* of a query. We then cut the resulting path(s) at all steps and store those 'cuts'[6] in collections of inverted queries on the various types (Contexts, Roles, Properties). On storing them, we must use the same keys we use when we retrieve them. How do we compute these keys in compile time?

Let's step back for a moment and contemplate our predicament. What we need to do is to add the cuts to the DomeinFile in such a way that we can find the relevant ones when we have a Delta on our hands. The strategy to follow depends on the Delta and the first step of the inverted query.

[6] Actually, it is more like 'kinking' the query path at all steps. So a query with n steps results in n-1 kinked queries. Each consists of two parts: a part going backwards (against the direction of the original query) and forwards (so it is the rest of the original query from that point). However, in this text I just call them 'cuts' and you can think of it of all subpaths of the inverted query leading to its end.

Keys for the Value2Role step

A Property query calculates, for a given role, the values of a particular PropertyType. When we invert that query it starts with the values. However, values are not stored as indexable entities in the PDR. Instead, we skip the first step of the inverted query and start with the role instance. Hence we can derive from a RolePropertyDelta on the instance level a 'PropertyInRole' element on the type level: the combination of a EnumeratedRoleType and a EnumeratedPropertyType. When we store a particular cut on a Property, we use that as a key to file it under. Actually, the EnumeratedPropertyType part will not discriminate anything, because it will always be equal to the type of the Property we've stored the cuts in. So we can make do with just the EnumeratedRoleType part.

It enables us to distinguish a Property in its lexical context (let's say A) from that same Property in some Aspect role context (say, B). Suppose that for B we have a CalculatedProperty that computes the average of the Property's values. Now, when that value set changes on an instance of B, we should recompute the average. But imagine that the value of the property changes on an instance of A. If we had not filed the cut under the key B, we were doomed to recompute the cut for A's instance, too. As a consequence, we might end up informing some peers that have nothing to do with that change.

In compile time, we have a QueryFunctionDescription whose Range is an ADT RoleInContext (it is the inversion of a PropertyGetter QueryFunctionDescription, whose *domain* is an ADT RoleInContext. From that range we take all EnumeratedRoleTypes and store the cut on the Property under each of those as key. This we do for all EnumeratedPropertyTypes we find in the domain of the QueryFunctionDescription.

Keys for the role step

When the inverted query approaches a role from the other direction (from the context, using a role step), we have a similar story. The `QueryFunctionDescription` has a Range that is an `ADT RoleInContext`. Since the domain can be a complex `ADT` holding many `ContextTypes`, the range can be a complex `ADT` holding many `RoleInContexts`.

For each of these `RoleInContexts`, we store the cut on the `ContextType` under the `EnumeratedRoleType`.

Keys for the context step

When we traverse the connection between role and context in that direction, we have a QueryFunctionDescription with a Range whose value is an ADT ContextType. It's Domain is an ADT RoleInContext. For each of the RoleInContexts we find in that Domain, we store the cut in its EnumeratedRole, under its ContextType.

Keys for fills and filledBy steps

A RoleBindingDelta describes a particular *segment* of the role- and context instances network. Moving into type space, we can project that instance segment on a particular *segment of RoleInContext* nodes. We'll call that pair the *TypeLevelSegment* and its first RoleInContext its *start* and the last its *end*.

Our task, then, runtime, is to find for a TypeLevelSegment the relevant cuts: the TypeLevelSegment is our *key*, on a conceptual level. So how do we find those cuts in the DomeinFile? How should we add cuts to the DomeinFile so we know the key will return the ones we look for?

A `QueryFunctionDescription` holds a domain and a range. These are *abstract datatypes* of `RoleInContext` - not a single segment. The simplest possible case would be a simple type (each consisting of a single `RoleInContext`) for both domain and range: then the queries' domain and range form a single TypeLevelSegment. Mapping the TypeLevelSegment derived from the `RoleBindingDelta` is easy in that simple case.

What if the domain of a particular cut consists of a SUM of two `RoleInContexts`? A moment's reflection learns us that if the start of the TypeLevelSegment is either of these two `RoleInContexts`, this cut should be evaluated. After all, the elements in a SUM represent alternatives. So now we can derive *two* TypeLevelSegments from the `QueryFunctionDescription`. Should our `RoleBindingDelta` map to either of them, we must evaluate the inverted query.

A PRODUCT of two `RoleInContexts` represents not alternatives, but composition. We should consider the product of two role instances to be a single role instance with combined properties. Again, if the start of the TypeLevelSegment we derive from the `RoleBindingDelta` is one of the `RoleInContexts` of the domain of the inverted query, we should evaluate that query (changing part of the composition is as good as changing the whole).

So we see, that as long as the start of the TypeLevelSegment derived from the `RoleBindingDelta` occurs somewhere in the abstract datatype of `RoleInContexts` that describes the domain of the inverted query, we should pick that query and re-evaluate it.

Or, formulated differently, for each `RoleInContext` occurring in the domain of the `QueryFunctionDescription`, we can construct a key from it with the RoleInContext of the range. The query should be stored under each key.

But wait. The *range* of the `QueryFunctionDescription` can be a SUM or PRODUCT, as well. Then what?

For the various `RoleInContexts` in the range, we can tell a story similar to what we said above about the domain. It's symmetrical. Given a single `RoleInContext` in the domain and several in the range, we should create a key with each of the latter combined with the first.

A problem arises when *both* domain and range have multiple `RoleInContexts`. Should we form the full Cartesian product of the `RoleInContexts` in both? It turns out we should not (see the example below): we'll generate far more keys than will ever turn up in runtime. Now this does not lead to semantically wrong results, but it is an efficiency issue not to be snuffed at (quadratic in complexity).

There is, however, a way to generate just the required keys. For this we re-run (during this process of storing inverted queries in the DomeinFile) the process of mapping the domain to the range when we compile a `fills` or `filledBy` step.

For the `filledBy` step this consists of looking up, for each RoleInContext in the Domain (of the cut), its binding (an ADT `RoleInContext`). Then, if a target context type has been specified in the query step, we replace the context in the `RoleInContexts` of that binding with the specified context[7].

For the `fills` step, the situation is somewhat simpler. The range of a fills step is, by construction, always a simple `RoleInContext ADT` whose `EnumeratedRoleType` is completely specified by the `fills`' step first parameter. If no context type is given (as a second parameter), we just use the `RoleInContext` as we find it as the endpoint and combine it with the `RoleInContexts` we find in the domain. Otherwise, we replace the context type in the end `RoleInContext`.

To sum up: we can construct, for each cut of an inverted query starting with the fills or filledBy step, a number of keys (TypeLevelSegments) we should store the cut under. Each TypeLevelSegment consists of two `RoleInContexts`; each `RoleInContext` consists of a `ContextType` and an `EnumeratedRoleType`.

We want to store the cuts on an `EnumeratedRole`. By convention we store it on the `EnumeratedRole` whose type is in the first (domain) `RoleInContext` of the compound key. We obviously don't need that first `EnumeratedRoleType` in the key that we store the cuts under. So we end up with a key compounded from the `ContextType` of the first `RoleInContext` and the two parts of the second `RoleInContext`.

Summary: sets of inverted queries

The table below gives, for several types, the subdivided sets of inverted queries that are stored on them and how to construct the keys to index them (in runtime) and to store them (in compile/type time).

Type	Member holding inverted queries	Key construction in runtime	Key construction in type time
Property	invertedQueries	EnumeratedRoleType	RoleType in Range
Context	invertedQueries	EnumeratedRoleType	RoleType in Range
EnumeratedRole	contextQueries	ContextType	ContextType in Range

[7] It requires us to store that target context in the QueryFunctionDescription.

	filledByInverted Queries	ContextType Filled – ContextType Filler – EnumeratedRoleType filler	ContextType Domain, ContextType Range, RoleType Range
	fillsInvertedQueries	ContextType Filler – ContextType Filled – EnumeratedRoleType filled	ContextType Domain, ContextType Range, RoleType Range

Comparison to Aspect Perspectives

In the discussion so far, we've restricted ourselves to the use of Aspects as the object role of a Perspective. However, we can have user roles in Aspect Contexts, too. Suppose we had added to Party a user role Giver, with a perspective on Present and its Price.

We could then cast Father as a specialization of Giver (and Mother too)[8]. Now there will be *a single* inverted query on Price. The perspective of Giver would have to be *contextualized* to Birthday Mother, which means that its object would be changed to Present Mother.

In short, the perspective of Giver is adapted and *added to Father's perspectives*. Only afterwards will the perspective's query be inverted. Consequently, it will be indexed by the Present Mother Enumerated role when it is stored with Price. Symmetrically, the same happens to Mother's perspectives and the inversion of the new perspective object will be indexed with the Present Father Enumerated role.

Price will then bear *three* inverted queries:

1. one indexed with Present (the Aspect object role in Party) for Giver (the Aspect subject role in Party),
2. one indexed with Present Mother, for the Calculated Father in Birthday Mother and
3. one indexed with Present Father, for the Calculated Mother in Birthday Father.

When the Price property of Present Mother changes, only one of these three queries will be followed to a context. This will turn up with an instance of Birthday Mother and we will then calculate Father in it and make sure the Price delta is sent to him.

Keys for aspects

We've shown how to construct keys in runtime from the types of the instances. The most complicated keys are derived from `RoleBindingDelta`s, consisting of two `ContextTypes` and one `RoleType`. But each of these types can be composed of Aspects. What are the consequences for the keys that should be constructed?

[8] Actually, we cannot do that in this model, as Father and Mother are calculated in the Birthday contexts and Calculated Roles cannot have Aspects. But the reasoning would apply for Enumerated Roles.

Each instance has a collection of types. A key is a triplet; should we construct all triplets from the type collections? At least we know that all relevant keys are in this product, but there are constraints on the combinations:

1. Aspect Roles and Aspect Contexts of a `Role`- and `ContextType` pair cannot be freely combined. We must treat `Role`- and `ContextTypes` as fixed pairs, where the `RoleType` is the lexically embedded in precisely one `ContextType`.
2. The possible fillers of a role are constrained by the possible fillers of its Aspects. If we say that Driver is constrained to be filled by a Human, we cannot have a role ArmouredCarDriver that uses Driver as Aspect but should be filled by a Robot (unless we let Robot have Human as Aspect).

As a consequence, on constructing triplet keys for queries starting on the `filledBy` step we can work as follows:

1. We start with the Filled role type;
2. We add to that all its Aspects;
3. For each of these we construct a key that consists of:
    a. Its context type;
    b. The type of its Filler;
    c. The type of the lexical context of that Filler.

So, from a single RoleBindingDelta, we construct as many triplet keys for the `filledBy` step as there are types of the filled role instance.

For the fills step we follow the same procedure (working from the Filled role type), but we construct the key by leaving out the type of the Filler role instance instead of that of the Filled role instance.

Handling complex filler types

But we're not done yet. A role may have a complex filler type, expressed as an `ADT RoleInContext`. Let's consider the role Parent to be filled by either Father or Mother, so its filler type is SUM Father Mother. Clearly, we can derive *two* keys from that filler type.

So our final algorithm is:

1. Start with the Filled role type;
2. We add to that all its Aspects, forming the set of FilledTypes;
3. For each element FilledType of FilledTypes we construct a number of keys, by
    a. Listing all of the EnumeratedRoleTypes in FilledType's Filler type;
    b. for each of those Filler types we construct for the `filledBy` step a single key from
        i. FilledType's context type;
        ii. Filler;
        iii. The type of the lexical context of Filler.

c. And this is how we construct keys for the fills step instead. Take
    i. FilledType’s context type;
    ii. FilledType;
    iii. The type of the lexical context of Filler.