TMQL Introduction

Robert Barta rho@bond.edu.au

This introduction is a tour de force through TMQL, the upcoming Topic Map query language.

Introduction

Work in TMQL has begun long ago, having been kicked off by a number of proposals. After quite some background work—actually a prototype has been implemented and a lot of formal research regarding optimization has been done), the editors have produced another official draft specification.

We assume that your are familiar with Topic Maps per se and how to create maps. The following TMQL preview builds on an existing presentation, but it breaks somewhat with the TM tradition to use 'operas' as running use case. Instead, we will use the more general theme 'music', and will assume that we have a TM data store which contains various albums, (female or male) musicians and music groups (which are all artists and have persons as members). All these topics are connected via various associations, such as `is-produced-by` or `is-part-of`.

As TMQL also can be used to generate Topic Map content, obviously a notation for this has to be chosen. As we speak, uhm write, the exact form of CTM, the compact topic map notation is being defined. We here use AsTMa= 2.0 for the time being.

Setting Off

If you have used SQL before, then you will not be completely puzzled by the following query:

```
select $album
where
   is-produced-by (production: $album, producer: tom-waits)
```

This query (technically a query expression) will return all albums where Tom Waits is known to be a producer. `tom-waits` is an identifier of a topic which we happen to know to uniquely pinpoint that topic about that person in the map we query. The query processor will try to find an association of type `is-produced-by` and will check whether the topic `tom-waits` is playing the role `producer` there. If so, it will bind the variable `$album` to the topic playing the role `production` in that very same association. It will so work through the whole map and will collect all these variable bindings. Because we asked so in the select clause, the query processor will return a list of the bindings for that variable. What exactly we will get returned into the application, we will discuss a bit later.

If we wanted to make the query more watertight to return albums only (and not something else produced), then we will have to add another constraint to the WHERE clause:

```
select $album
where
   is-produced-by (production: $album, producer: tom-waits) &
   $album isa album
```

The special binary predicate `isa` checks now additionally whether the thing we have bound to `$album` is also an instance of the class `album`, at least according to the map we query. It is worth noting, that `isa` honors the (transitive) subclass-superclass relationship, i.e. if one album were an instance of `compilation`, and `compilation`, in turn, were a (direct or indirect) subclass of `album`, then also these topics would be successfully bound.

If we are not fixated on Tom Waits and would instead want a list of all albums together with their producers, we can extend the wishlist in the SELECT clause:

```
select $album, $producer
where
   is-produced-by (production: $album, producer: $producer) &
   $album isa album
```

Again, the processor would walk through the whole map, will find all associations of the given type and will bind the playing topics to their respective variables. One particular set of bindings now consists of a pair (tuple of two components) of bindings, each for one variable; all these pairs are collected in a list which is then eventually returned.

Controlling What is Returned

In the SELECT clauses we have used so far, we asked for whole topics. Any query processor will hand over a complete topic data structure (most probably according to TMDM) into the application. If an application were interested, say, only in the name of such a topic, it would have to use some API outside the scope of TMQL to navigate to that name.

To let the TMQL processor do the work, we can tweak the SELECT clause by adding a path expression:

```
select $album / name
where
   $album isa album
```

Now the processor will do the navigation for us, as we requested with `/ name` to find all the names for the any thing bound to `$album`. It also does here something not obvious: It not only finds the name characteristics, i.e. the structures holding the string value for the name together with a scope and a type; it also reduces such a structure into its string value, which is exactly what the calling application will see.

This process of converting a characteristics (be it a name or an occurrence) into a string is called atomification, as literals as numbers, strings, etc. are referred to as `atoms`. In most cases, this is highly convenient, in others this is not.

Generally, a topic can have any number of names—all with different type or scope—, so we actually would get a whole list of names for each individual album. If we were not interested in all names, but only in those in, say, english (`en`), then we want to filter the characteristic first based on the scope and only then atomify the name:

```
select $album / name [ @ en ]
where
   $album isa album
```

If the processor would stupidly atomify all names to strings immediately, then the scope information would be lost, so filtering according to scope would not be possible. This then is not what happens. In fact, the processor will postpone the atomification until the end of the path expression. The precise rules are somewhat contrived. Later we will also see that we can also completely suppress this behavior, if it does not suit our purposes.

Path expressions are also a convenient way to impose a sorting order on the list of tuples we return:

```
select $album, $producer
where
   is-produced-by (production: $album, producer: $producer)
order by
   $album / name [ @ en ]
```

That way we get albums and their producer, but the whole list becomes a sequence sorted according to the english album title.

The ordering can also include more than one ordering criterion, like in

```
select $album, $producer
where
   is-produced-by (production: $album, producer: $producer)
order by
   $producer / name [ @ en ] desc, $album
```

Here we first sort the list of topic pairs according to the name of the producer. For demonstration only we choose descending ordering. More importantly though, for one specific producer name (in the english scope) we sort the sublist containing different albums according to the album's identifier. This may not by itself overly useful, but at least it takes care that the whole returned list always appears in the same order if we keep repeating the same query.

As you would expect, TMQL makes it possible to make the list of returned tuples unique and to select only slices out of the whole result set. It could look like this:

```
select $album
order by
   $album / name
unique offset 10 limit 20
```

Identifying Things

You may correctly argue that identifying topics with their (internal) map identifier (the TMDM model calls this source locators) is not an immensely robust idea if that identifier may change any second. Less brittle solutions involve to use either subject locators to directly address a subject, or subject indicators for indirect identification.

In the case, for instance, that there is a subject indicator, you can use that instead of an internal identifier:

```
select $album
where
   is-produced-by (production: $album, producer: http://www.u2.com/ ~)
```

The tilde ~ after the URL of an U2 web site tells TMQL to find a topic which uses this URL as subject indicator. This notation—first introduced by Steve Pepper— also allows to specify that URL is supposed to be interpreted as subject address:

```
select $album
where
   is-produced-by (production: $album, producer: $producer) &
```

```
   is-maintained  (maintainer: $producer, website: http://www.u2.com/ =)
```

This time we used the URL to identify a topic in the queried map which is using the URL as subject address, i.e. reifying the U2 web site itself. Via a `is-maintained` association we connect that to one (or more) producer(s) and these with the albums we are looking for.

Controlling Variable Bindings

As we have seen above, variables can be bound to values. Used naively, this can lead to incorrect queries. As an example let us find all two albums which share the same producer. In a first attempt we write:

```
select $album1, $album2
where
   is-produced-by ($album1: production, $producer: producer) &
   is-produced-by ($album2: production, $producer: producer)
```

If you have worked with declarative languages before, you may immediately spot the problem: For the TMQL processor `$album1` and `$album2` are completely different variables; the variables might be bound to the same or to different value for the same producer, the processor does not care.

This does not work for us if we want different albums. The usual escape hatch is to have something like this:

```
select $album1, $album2
where
   is-produced-by ($album1: production, $producer: producer) &
   is-produced-by ($album2: production, $producer: producer) &
   $album1 != $album2
```

Not only is this ugly as hell, in 100% - ε of all cases developers will forget to add this (I know I will). And it does not look too good if you have to compare three or more such variables.

TMQL has a rather eccentric way to fine-control when variables are allowed to match anything or when they must be bound to something different:

```
select $album, $album'
where
   is-produced-by ($album : production, $producer: producer) &
   is-produced-by ($album': production, $producer: producer)
```

Now we have used two variables which only differ in their name by the number of primes (') appended. TMQL treats them as two distinct variables, but with the additional semantics that— within one and the same binding—they cannot be bound to the same value.

There is no limit to the number of primes, so should we—by a bizarre twist of fate or customer requirements—need three different albums, this can be achieved nicely:

```
select $album, $album', $album''
where
   is-produced-by ($album  : production, $producer: producer) &
   is-produced-by ($album' : production, $producer: producer) &
   is-produced-by ($album'': production, $producer: producer)
```

Association Templates

In the queries so far we made use of association templates. Writing inside a query

```
   is-produced-by (production : $album, producer: $producer)
```

makes the processor try to find matching associations in the queried map. Such associations must be of type `is-produced-by` and must have exactly two roles, one for `production` and one for `producer`. If an association in the map has a third role, say, `location`, to capture where an album has been produced, then such association would never match the template.

To allow for such associations with additional roles to match, TMQL allows to append an ellipsis:

```
select $album
where
   is-produced-by (production : $album, producer: $whoever, ...)
```

Association templates actually have more implicit meaning than is obvious at first sight. If, for example, the map contained an association of type `is-remastered-by` which also connects an album with a producer and `is-remastered-by` is a subtype of `is-produced-by`, then also such associations would match the template.

Honoring subclassing also applies to roles. Had we in our queried map an association of type `is-remastered-by`, but the role (type) for the album is not `production`, but the subclass `remastering`, such association would also match the association template.

If you would not care about the role, you can use a wildcard there:

```
select $album
where
   is-produced-by (production : $album, * : $whoever, ...)
```

The * is equivalent with `xtm:subject`. Obviously, everything in the map is an instance of a subject. Using such wildcards may be convient; it may also be walking on thin ice as processors have less knowledge to start with. This may make an impact on the performance.

Query Context

You may have wondered how the query processor knows which map to query. One way is that the map itself is hard-coded into the query expression itself:

```
select $album
from file:mymap.xtm ~ >>
where
   ...
```

Assuming that we have stored an XTM topic map in the file `mymap.xtm`, the path expression `file:mymap.xtm ~ >>` makes several steps: First, we have implicitely indicated a subject, namely the map stored in the file. In a TM sense, we also temporarily create a topic for the map. Then we zoom into the map, i.e. ask the processor to parse the file, interpret the content as a collection of topics and assocation; and, finally, we switch our focus of attention to that map, so that the rest of the query expression is interpreted in the context of this map.

While hard-coding such information into a query may make sense in some cases (query processors may want to analyse the map before query time in order to optimize), generally one would like to pass this map information at query time to the TMQL processor. This is done via a mechanism we already have seen: variable bindings where a value (string, integer, but also whole maps) are bound to variables. Applications will build these binding sets and pass them somehow to the processor.

For our example, our application could have already loaded the map into memory, possibly having done some modifications already to it. When that map is bound to, say, `%m`, then inside the query we can refer to this variable, blindly trusting that it contains a map:

```
select $album
from %m
where
   ...
```

As this scheme—passing in a map to be used in a FROM clause— is expected to be used often, it can be abridged. First, we can make use of the special variable `%_`; whatever map is assigned to it, that map will be the one queried. Which implies that the clause `from %_` is default anyway and can be omitted.

The query context can contain other variables as well. Slightly paraphrasing an earlier query, we could parameterize it with a URL which we additionally pass in bound to `$url`:

```
select $album
where
   is-produced-by (production: $album, producer: $url ~)
```

Path Expressions

The textual overhead of the SQLish style which we have used so far may not be convenient if queries are trivial. Especially for web applications where pages have to be filled with lots of content from a TM backend a much shorter notation is more adequate.

To return all albums from the map bound to `%_` we can simply write

```
// album
```

If we need the english names only, then

```
// album / name [ @ en ]
```

will do just fine.

We can also, for instance, reformulate an earlier query which returned only the english names of Tom Waits albums:

```
// album [ . <- production [ * is-produced-by ] -> producer = tom-waits ] / name [ @ en ]
```

The processor will again start off with all albums and will subject each of them to a test provided by the filter condition in the first `[]` group. That will effectively test whether Tom Waits is one of the producers. This is accomplished by taking each album (the dot `.` symbolizes the current item), checking out all associations where this item is playing the role `production`, filtering these associations so that only those remain which are of type

`is-produced-by`; when following the role `producer` from these associations we end up with a list of topics playing that role. This list will be compared with the list on the right side of the = symbol. While in our case there is only one, and it is a constant anyway, this comparison is trivial and returns true if one of the producers happens to be the same topic as that one with the identifier `tom-waits`.

Only these albums where the condition is satisfied will be postprocessed in that the english name is selected from them. Implicitly, the string values are taken.

Chocolate, Vanilla, Caramel

The different language flavours, select and path expressions, can—as we have already seen— be mixed. Not so obvious is the fact that both styles are (almost) equivalent in terms of expressitivity; every select query expression can in fact be transformed into an equivalent path expression.

As path expressions cannot introduce new variables, they can become quite contrived when they get more complex.It is up to the developer to choose the most appropriate combination on a case by case basis.

Both these styles allow to return sequences of tuples of things into the application. This may be exactly along your line of thinking in most cases, but it does not help tremendously if you want to comfortably embed the query results into one of these shiny XML applications servers. To avoid that developers have to write their own template engines, TMQL allows to create XML content using a third flavour, which—you may have guessed it—is otherwise equivalent to the other styles. This flavour, FLWR, is inspired by XQuery and uses RETURN clauses to specify the output:

```
return
   <albums>{
     for $a in // album return
         <album>{$a / name [@ en ]}</album>
   }
   </albums>
```

The return value here will create one XML 'document' as data structure (implementations will probably choose DOM for that) with a root element `<albums>`. Nested into that will be all albums in the map. The way this is achieved is by iterating over albums in a FOR loop. It uses a path expression `// album` to compute first all instances of albums in our map. Each such album is bound to the iteration variable `$a` and with such a new binding, the body of the loop is evaluated.

Such body is defined by a nested RETURN clause. It contains an element `<album>` which wraps only text content which we specify with an embedded TMQL path expression `$a / name [@ en ]`. Like in XQuery, XML content and query text is separated using {} brackets. Since the processor knows that here text has to be, it will implicitly take the string value of that basename.

Using Exist and All Quantification

On some occasions you will have to test whether particular things exists in a map or whether all things in certain set have a particular property. For illustration, let us ask for all music groups in our map which have at least one female group member

```
for $group in // group
where
    some $person in $group <- whole -> member
         satisfy
             $person isa female
return
    ($group)
```

While we iterate over all groups in the map, we find for each such group all members using the path expression `$group <- whole -> member`. If only one satisfies the condition that it is an instance of `female` then the existential SOME clause is satisfied.

Conversely, we might be interested to find all boy groups, well, at least those groups where all members are male:

```
for $group in // group
where
    every $person in $group <- whole -> member
         satisfies
             $person isa male
return
    ($group)
```

Predefined Datatypes and Functions

Since the latests TMDM drafts include features to store arbitrary data (and not just text as in the original XTM-based standard drafts), TMQL has provisions for data. The obvious first thing to support is to denote constants; here TMQL borrows (stealing is such an ugly word) from RDF(S) notations:

```
select $person / name
where
   $person / age > "18"^^xsd:integer
```

This query will select names for everyone older than 18.

Now, writing integers, floats, or even dates and strings in this explicit form is somewhat clumsy. Therefore TMQL has adopted a few of these primitive types, specifically integers, decimals, dates, URIs and strings, all imported from the XSD (XML schema data types) namespace. Constants of these types can be written without the explicit typing fluff:

```
select $person / name
where
       $person / age  >  1           &
       $person / born >= 1962-06-06  &
   not ( $person / name     = "Bill Gates" |
         $person / homepage = http://he.is.so.good.to.us/ )
```

As a consequence, also necessary functions for these data types have been imported into TMQL and are part of the predefined environment. This includes also the comparison functions which we have used above. Actually, when writing a comparison like `$person / age > 18`, several things happen: first we extract the `age` occurrence from a person. This characteristic will then silently atomified to its value. If that value is already an integer, then it will be used as-is. Otherwise TMQL will try to convert it, which may only work if the value is a string with only digits in it. Then the two integers will be compared using the predefined function `op:numeric-greater-than`.

There is also another twist here: You may have noted that a path expression like `$person / born` may return more than one characteristic; or none, depending how many `born` occurrences actually exist for the given topic. The interpretation TMQL chooses here is exists semantics. TMQL will translate this to a more explicit

```
some $_16523 in $person / born
    satisfy
        op:dayTimeDuration-greater-than ($_16523, 1962-06-06)
```

whereby the variable `$_16523` is internally generated. If the processor had access to a description of the queried map (the ontology) and if it could learn that there is always exactly one `born` property for every person, an optimizer could collapse the above SOME clause.

Prefixes

Using whole URIs to identify topics (actually the subjects they stand for) certainly clutters queries. Using prefixes for namespace is a comfortable way to shorten queries considerably, so TMQL also uses this mechanism; albeit, not in a syntactic way, like in XML, or RDF, but more appropriately here, in a semantic sense.

While the `xtm` prefix is actually one of the predefined ones, it could be redefined like this:

```
"""
xtm isa tmql:ontology ~ http://www.topicmaps.org/xtm/1.0/#
"""
select $thing
   $thing isa xtm:topic
```

The part in between the `"""` is a topic map, the so-called meta map; it can contain any topics and associations, more about that later. In this case we defined only a single topic `xtm` which obviously is an instance of an `tmql:ontology`. Which ontology we mean, we indicate with a subject identifier `http://www.topicmaps.org/xtm/1.0/#`.

For the TMQL processor this mean two things: First, it will try to understand the contents of that ontology (or vocabulary). This can be done by either downloading the file, parsing it and remembering the concepts defined therein. It can also mean that the processor already knows about that URI and therefore vocabulary it stands for. Here it does not really make a difference whether the vocabulary is defined as topic map or via, say, an OWL ontology. The second thing the processor does, is that the topic source identifier (`xtm`) can be used as prefix within the query which directly follows the map. When the processor encounters `xtm:topic`, it will find the prefix `xtm` and will expand the QName to `http://www.topicmaps.org/xtm/1.0/#topic`; that URI is then interpreted as subject identifier in the queried map. That, of course, only makes sense to be used if such a subject is covered there.

If, in contrast, we would import an ontology which has no relation to the map to be queried, then such a prefix is useless at first. But let us look at the following query fragment:

```
"""
opera isa tmql:ontology ~ http://www.ope-r-us.org/opera.ltm
"""
select $opera
from %_ + opera >>
   ...
```

Similar to above, but this time using LTM as notation, we have imported an ontology and have bound it to the

prefix `opera`. Whatever that ontology contains—we assume it to be information about operas, some taxonomy perhaps—, we will use that vocabulary and merge it into our queried map. The is done by refering first to our current map `%_`, enriching it using `+` by the map which can be found when zooming into the topic `opera`.

Inside the query body, the `opera` prefixed can the be naturally used:

```
....
   $opera isa opera:opera &
   written-by (opus: $opera, author: tom-waits)
```

Finally, what if an ontology exists, but is not in Topic Map format but, say, in OWL? TMQL processors are free to covert from other notations and paradigms into a topic map:

```
"""
wine isa tmql:ontology ~ http://www.w3.org/2001/sw/WebOnt/guide-src/wine.rdf
"""
select $winery
from %_ + wine >>
where
    $winery is-a wine:Winery
```

User-defined Functions

Once queries get more complex, they have to be organized and engineered. One vehicle in this process is to compose functions. These can compute values from provided parameters and—semantically speaking—define a functional relationship between values and other things.

We can declare a function in the meta map; there it is just a topic of a certain type:

```
"""
nr-albums isa tmql:function
description: this function computes the number of albums a certain person has produced
return: fn:length (// albums [ . <- opus -> author = $person ])

"""
```

The TMQL processor will register this topic as function then. Since it is a normal topic, we can attach all sorts of relevant (or irrelevant) information. One important part is the body, where we can use TMQL to define what should be returned. Here it is a one-liner, a path expression, but any TMQL style is admissible here. From the expression and the one free variable `$person`, the processor will also learn that there is one parameter `person` to that function.

Inside the query we can make use of the function:

```
select $p / name, nr-albums (person: $p)
where
   $p isa person
```

Note that we have to associate a current person (stored in `$p`) with the formal parameter `person` to passed it into the function.

Any number of parameters can be introduced this way. What can also be done is to provide default values for function parameters, if the caller does not provide one. The following function `top-albums` finds the most popular albums for a given producer:

```
"""
top-albums isa tmql:function
description:
return: """
   select $album
   where
       $album isa album &
       is-produced-by (producer: $person, opus: $album)
   order by $album / popularity desc
   limit $limit || 5
"""

"""
```

While the function uses 3 different variables, only `$person` and `$limit` are free i.e. not quantified by the query itself. The expression `$limit || 5` is only a shortcut for the more elaborated `if $limit then $limit else 5`, it uses the parameter if that is defined and uses 5 otherwise.

If the function is invoked with all parameters, these then will be used:

```
select $person / name, top-albums (person: $person, limit: 3) / name
where
   $person isa person
```

As a consequence, we will receive a list of person names and album names; at most 3 albums for one person. Alternatively, we can omit the limit, so that the default 5 is used.

The fact that functions are topics, allows a further variation of the theme. A processor may allow a developer to use a language other than TMQL, say, Python:

```
"""
ctime isa tmql:function
return @ python: """
    from datetime import ctime
    return ctime()
"""

"""
```

All we needed to do was to set the scope accordingly. Needless to say, that this is an excellent extension mechanism, but only if the function itself is short to be directly included. For bigger external libraries it is probably more manageable if the source code lives externally:

```
"""
math isa tmql:ontology ~ file:/usr/local/math/api.atm
"""
```

which brings us back to importing whole ontologies. The only difference to be before is that the `api.atm` file contains the functions, all topics of type `tmql:function`:

```
...
pow  isa tmql:function
...
sqrt isa tmql:function
...
```

To use these functions inside the query, we have to prefix them: `math:pow` and `math:sqrt`.

What if the functionality should or cannot be encoded in source code as part of a topic map? Also this case is covered as ontological import:

```
"""
math isa tmql:ontology ~ http://maths.is.great/
"""
```

As before, we used a URI to indicate the ontology. Unlike before, the implementation of the processor allows us to bind a, say, Python library to exactly this URI:

```
import math

proc = TMQL.processor()
proc.registerNS ('http://maths.is.great/', math)
```

As soon as it recognizes the URI as one registered namespace, it will not try to dereference the URI, but will rather load the math library. From then on, everything works as before.

And there is a last benefit for registering functions in the meta map: introspection. With it, it is possible to actually query the meta map itself to learn about the functions:

```
select $f / description
from %__
where
    $f isa tmql:function
```

The special variable `%__` not only names the meta map; inside a FROM clause it also causes the processor to switch its focus on the meta map and to interpret everything following relative to it. The rest is history, as they say.

Association Predicates

Predicates are tests, so they either return `true` or `false` depending on whether the claim they represent is verifyable or not. In this sense, predicates are specialized function. And since functions can be declared in the meta map, so can predicates.

As example let us consider that our queried map contains associations of type `has-created` where we have recorded which artist (single or group) has created which album, such as "U2 as creator has created an opus 'Rattle and Hum'", or "'Leonard Cohen as creator has created an opus 'Various Positions'".

It lies in the nature of associations of type `has-created` that if a group has created something, then we might also deduce that every member of the group also has created that thing. Naturally, we do not want to change our map and add all this redundant information to it. Instead we would like add a rule to the purpose of "if an individual

belongs to a group and the group has created something, so has the individual".

```
"""
has-created-indirect isa tmql:predicate
where: """
     has-created (creator: $creator, opus: $opus)
     |
     is-part-of  (member : $creator, whole: $group) &
     has-created (creator: $group,   opus : $opus)
"""
```

Obviously, this predicate is nothing else than a boolean condition. Different to functions, though, is that there are no parameters that a caller must provide. How the predicate is operating depends on its use:

```
select $person
where
     $person isa person &
     $thing  isa *      &
     has-created-indirect (creator: $person, opus: $thing)
```

Here we iterate over all persons and all things in the map. For each of these pairings we then invoke the predicate whereby we pass one person as `creator` and one thing as `opus`. In the predicate now the only unbound variable is `$group`. The processor will now try to evaluate the predicate whereby it will keep `$creator` and `$opus` fixed, but will vary the only unbound variable `$group` by letting it iterate over all possible values.

The obvious benefit of such rules is to keep redundancy in maps itself low, while not burdening the query itself; it would have been completely equivalent to hard-code it there:

```
select $person
where
     $person isa person &
     $thing  isa *      &
     (
        has-created (creator: $creator, opus: $opus)
        |
        is-part-of  (member : $creator, whole: $group) &
        has-created (creator: $group,   opus : $opus)
     )
```

Nested Queries

Nesting of query expressions is quite natural and could potentially be used in quite a few places. To demonstrate how much the language can be stretched, here an slightly obstruse example to use nesting within a SELECT clause.

To lists all groups and their members in the map, the SELECT clause may contain a whole subquery:

```
select $group / name,
       " members are: " + fn:string-join (
                               fn:tuple ({
                                   select $person / name
                                   where
                                       is-part-of  (member: $person, whole: $group)
                                   }), ",")
where $group isa group
```

The query is actually initiated by finding first all instances of the concept `group`. For each of them, the expressions in the SELECT clause are evaluated. The first column there is always a string containing one group name. The second column is also a string, but one that is concatenated from a constant string and the result of a nested query expression.

For a given group we will there find all members and extract their name(s). In any case, the subquery will return a table with a single column, that with the names. To convert this into a list of things, we use the function `tmql:tuple`; only then we can concatenate the individual strings with commas separated using `fn:string-join`.

Generating Topic Maps

We have already seen that TMQL can generate lists (of tuples) and XML content. But with TMQL we can also generate topic maps, using information from the queried map, provided we use the FLWR style.

In the following example we iterate over all albums in our map and create a new map, one which only contains the albums and one of their names. All other album information (characteristics or association involvements) is ignored. Instead we add one `homepage` characteristic and one—admittedly absurd—new assocation where we claim that Jessica Simpson has created that album:

```
for $a in // album
return """
```

```
    {id ($a) + "-new"}
    {$a/name}
    homepage: http://music-r-us.com/{id ($a)}

    (has-created)
    creator: jessica-simpson
    opus   : {$a}

"""
```

The RETURN clause now contains a topic map, or—to be more precise—a template to generate one. For each album we have found, this template is expanded, simply by evaluating all expressions in curly brackets.

The first block follows the syntax to create a topic, so the very first thing to appear here is the internal identifier of the topic. That is computed by looking up the internal identifier of the current album and by appending it with `-new`.

The next line again contains a TMQL expression, this time one which computes all `name` characteristics. Since at this position in the notation characteristics can be declared, all these names become part of the newly generated topic. The last line of the topic declaration will take care that an occurrence of type `homepage` will be added to the topic. The value is a URI which is static except from the final part which is provided by the albums internal identifier.

The second block creates one association per album, claiming the unbelievable.

As this template is expanded for every album, the individual expansions are all merged into one, possibly big, topic map. As we did not say much about Jessica Simpson we might want to do so, and combine this information with our result map:

```
return """

   jessica-simpson isa person
   bn: Jessica Simpson
   quote: "I am an artist."

{
 # here our original query goes
}
"""
```