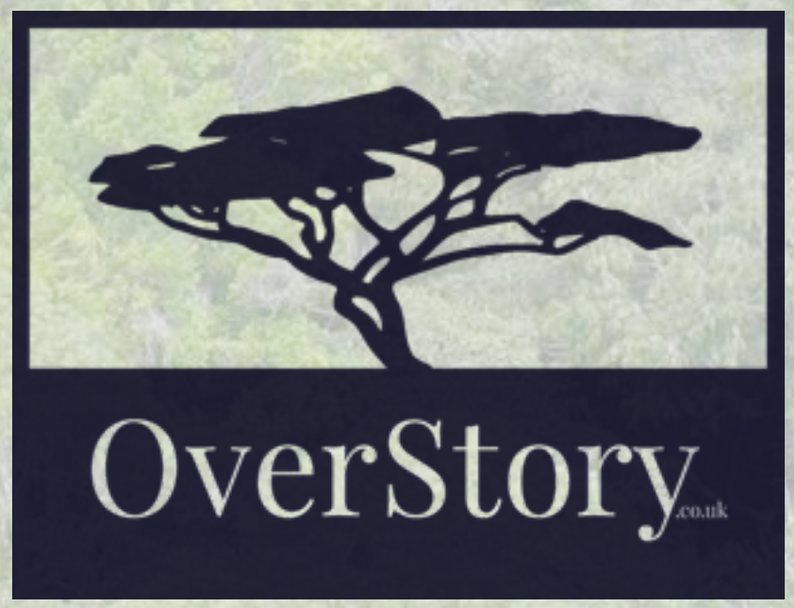
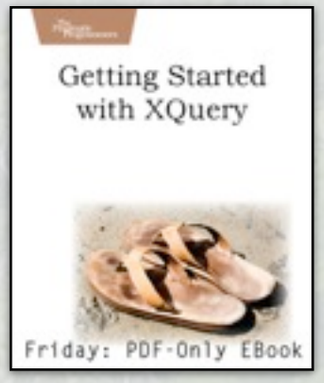
Stupid XQuery Tricks

Ron Hitchens
Principal Consultant, OverStory Ltd
ron@overstory.co.uk



Who Am I?

- Former Lead Engineer at MarkLogic (5 years)
- Implemented the XQuery 1.0/1.0-ml parser in ML
- Wrote an XQuery book for The Prags
- Written lots of XQuery over the years
  - (Some of it quite stupid)



Stupid XQuery Tricks

XPath Fancy Footwork

Subvert The Established Order

Fun With Functions

Check the Map

Potpourri

XPath Fancy Footwork

- Predicates
- Final steps
- Simplify FLWORs

Pick an alternative from a sequence

- Supply defaults for possibly empty values
- Select highest priority item from a sorted list
- Pick a lottery winner

Supply default values

```
declare variable $DEFAULT-TYPE as xs:string := "article";

declare function process-content (
  $pub as element(pub),
  $parent as element(pub))
{
  let $content-type :=
  (
    fn:data ($pub/@type),
    fn:data ($parent/@type),
    $DEFAULT-TYPE
  )[1]

  return handle-content-for-type ($content-type, $pub, $parent)
};

...
```

Choose highest priority item

```
declare variable $TODO-COLL := "todo";

(: Extract meta element from oldest doc in “todo” collection
let $meta :=
  (
    for $doc in fn:collection($TODO-COLL)
    order by xs:date ($doc/root/@submitted-date) ascending
    return $doc/root/meta
  )[1]

return (
  process-submission ($meta),
  xdmp:document-remove-collections (
    xdmp:node-uri ($meta), $TODO-COLL)
)
```

Pick a lottery winner

```
let $count := xdmp:estimate (fn:doc()/contestant)
let $winning-number := (xdmp:random() mod $count) + 1
let $winner := fn:doc()/contestant[$winning-number]

return
bestow-fortune ($winner)
```

XPath final steps

XPath final step can be a function call or constructor

Can only apply to a sequence of nodes, not atomic types

Eliminate FLWORs

```
(: FLWOR form :)
for $i in $some-sequence/some/path
return some-function ($i)

(: XPath form :)
$some-sequence/some/path/some-function(.)
```

Can return nodes or atomic values, but not both

Execution order may be different than FLWOR

XPath final step can be a node constructor

Wrap nodes in a new constructed node

```
declare variable $animals :=
  <animals>
    <animal kind="mammal">aardvark</animal>
    <animal kind="bird">goose</animal>
    <animal kind="insect">locust</animal>
    <animal kind="bird">raven</animal>
    <animal kind="mammal">zebra</animal>
  </animals>;

<creatures>{
  $animals/animal/<creature>{ . }</creature>  (: wrap animal :)
}</creatures>
```

Resulting sequence

Note that sequence order has **not** been preserved

```
<creatures>
  <creature>
    <animal kind="mammal">zebra</animal>
  </creature>
  <creature>
    <animal kind="bird">goose</animal>
  </creature>
  <creature>
    <animal kind="mammal">aardvark</animal>
  </creature>
  <creature>
    <animal kind="insect">locust</animal>
  </creature>
  <creature>
    <animal kind="bird">raven</animal>
  </creature>
</creatures>
```

Can also be a computed constructor

Rename elements dynamically

```
declare variable $animals :=
  <animals>
    <animal kind="mammal">aardvark</animal>
    <animal kind="bird">goose</animal>
    <animal kind="insect">locust</animal>
    <animal kind="bird">raven</animal>
    <animal kind="mammal">zebra</animal>
  </animals>;

<creatures>{
  $seq/animal/element { @kind } { node() }
}</creatures>
```

Can also be a computed constructor

Rename elements dynamically

```
declare variable $animals :=
  <animals>
    <animal kind="mammal">aardvark</animal>
    <animal kind="bird">goose</animal>
    <animal kind="insect">locust</animal>
    <animal kind="bird">raven</animal>
    <animal kind="mammal">zebra</animal>
  </animals>;

<creatures>{
  $seq/animal/element { @kind } { node() }
}</creatures>
```

```
<creatures>
  <bird>raven</bird>
  <bird>goose</bird>
  <mammal>zebra</mamma
  <mammal>aardvark</ma
  <insect>locust</inse
</creatures>
```

XPath Predicates can simplify FLWORS

- Predicate in for expr is roughly like a where clause
- Predicates are somewhat more likely to be optimized

```
(: With a where clause :)
for $user in fn:doc()/user
where ($user/sex = “F”) and ($user/age lt 6)
  and ($user/color = “pink”)
return princess ($user)

(: XPath form :)
for $user in fn:doc()/user[sex = “F”][age lt 6][color = “pink”]
return princess ($user)

(: XPath final step as function form, no FLWOR at all :)
fn:doc()/user[sex = “F”][age lt 6][color = “pink”]/princess(.)
```

Subvert The Established Order

- Random sort order
- Dynamic sort order

Random sort order

```
declare variable $beasts :=
  ( "lion", "tiger", "bear", "elephant", "moose" );

declare function local:sort-beasts()
{
   for $beast in $beasts
   order by xdmp:random()
   return $beast
};
```

Random sort order

```
declare variable $beasts :=
  ( "lion", "tiger", "bear", "elephant", "moose" );

declare function local:sort-beasts()
{
   for $beast in $beasts
   order by xdmp:random()
   return $beast
};
```

```
local:sort-beasts()
=>
elephant
moose
bear
lion
tiger
```

Random sort order

```
declare variable $beasts :=
  ( "lion", "tiger", "bear", "elephant", "moose" );

declare function local:sort-beasts()
{
   for $beast in $beasts
   order by xdmp:random()
   return $beast
};
```

```
local:sort-beasts()
=>
bear
tiger
elephant
moose
lion
```

Random sort order

```
declare variable $beasts :=
  ( "lion", "tiger", "bear", "elephant", "moose" );

declare function local:sort-beasts()
{
   for $beast in $beasts
   order by xdmp:random()
   return $beast
};
```

```
local:sort-beasts()
=>
tiger
lion
bear
elephant
moose
```

Dynamic sort order

Use a flag to control the sort order of a FLWOR

```
declare variable $beasts :=
  ( "lion", "tiger", "bear", "elephant", "moose" );

declare function local:sort-beasts ($down as xs:boolean?)
{
   for $beast in $beasts
   order by
      $beast[fn:not($down)] ascending,
      $beast[$down] descending
   return $beast
};
```


Thanks: Rob Whitby

Dynamic sort order

Use a flag to control the sort order of a FLWOR

```
declare variable $beasts :=
  ( "lion", "tiger", "bear", "elephant", "moose" );

declare function local:sort-beasts ($down as xs:boolean?)
{
   for $beast in $beasts
   order by
      $beast[fn:not($down)] ascending,
      $beast[$down] descending
   return $beast
};
```

```
local:sort-beasts (fn:false())
=>
bear
elephant
lion
moose
tiger
```

 Thanks: Rob Whitby

Dynamic sort order

Use a flag to control the sort order of a FLWOR

```
declare variable $beasts :=
  ( "lion", "tiger", "bear", "elephant", "moose" );

declare function local:sort-beasts ($down as xs:boolean?)
{
   for $beast in $beasts
   order by
      $beast[fn:not($down)] ascending,
      $beast[$down] descending
   return $beast
};
```

```
local:sort-beasts (())
=>
bear
elephant
lion
moose
tiger
```


Thanks: Rob Whitby

Dynamic sort order

Use a flag to control the sort order of a FLWOR

```
declare variable $beasts :=
  ( "lion", "tiger", "bear", "elephant", "moose" );

declare function local:sort-beasts ($down as xs:boolean?)
{
   for $beast in $beasts
   order by
      $beast[fn:not($down)] ascending,
      $beast[$down] descending
   return $beast
};
```

```
local:sort-beasts (fn:true())
=>
tiger
moose
lion
elephant
bear
```


Thanks: Rob Whitby

Fun with Functions

- Function mapping
- Higher order functions

Function Mapping

- Use it to eliminate FLWORS (maybe several at once)
- Use in place of “if” statements
- Turning it off

How function mapping works

- It's a bit like XPath final steps, but applies to function parameters
- Attempting to pass a sequence to a function where it expects a singleton will generate an implicit FLWOR
  - It "maps" the function to each item in the sequence
- Passing an empty sequence iterates zero times

Example

```
declare variable $beasts := ( "lion", "tiger", "bear");

declare function local:tame ($beast)
{
    fn:concat ("Nice ", $beast)
};

local:tame ($beasts)
```

Example

```
declare variable $beasts := ( "lion", "tiger", "bear");

declare function local:tame ($beast)
{
    fn:concat ("Nice ", $beast)
};

local:tame ($beasts)
```

<== Implicit type is item()*

```
1.0-ml] XDMP-ARGTYPE: (err:XPTY0004)
  fn:concat("Nice ", ("lion", "tiger", "bear")) -- arg2 is not
of type xs:anyAtomicType?
```

Example, singleton parameter

```
declare variable $beasts := ( "lion", "tiger", "bear");

declare function local:tame ($beast as xs:string)
{
    fn:concat ("Nice ", $beast)
};

local:tame ($beasts)
```

Example, singleton parameter

```
declare variable $beasts := ( "lion", "tiger", "bear");

declare function local:tame ($beast as xs:string)
{
    fn:concat ("Nice ", $beast)
};

local:tame ($beasts)
```

```
Nice lion
Nice tiger
Nice bear
```

Example, singleton parameter

```
declare variable $beasts := ( "lion", "tiger", "bear");

declare function local:tame ($beast as xs:string)
{
    fn:concat ("Nice ", $beast)
};

local:tame ($beasts)
```

```
Nice lion
Nice tiger
Nice bear
```

```
local:tame ($beasts)

behaves like:

for $i in $beasts return local:tame ($i)
```

Mapping applies to all singleton params

```
declare variable $beasts := ( "lion", "tiger", "bear");
declare variable $counts := ( 7, 1, 3 );

declare function local:tame ($beast as xs:string,
  $count as xs:int)
{
    if ($count = 1)
    then fn:concat ("Nice ", $beast)
    else fn:concat ($count, " ", $beast, "s, oh my!")
};

local:tame ($beasts, $counts)
```

Mapping applies to all singleton params

```
declare variable $beasts := ( "lion", "tiger", "bear");
declare variable $counts := ( 7, 1, 3 );

declare function local:tame ($beast as xs:string,
  $count as xs:int)
{
    if ($count = 1)
    then fn:concat ("Nice ", $beast)
    else fn:concat ($count, " ", $beast, "s, oh my!")
};

local:tame ($beasts, $counts)
```

```
7 lions, oh my!
Nice lion
3 lions, oh my!
7 tigers, oh my!
Nice tiger
3 tigers, oh my!
7 bears, oh my!
Nice bear
3 bears, oh my!
```

Use function mapping to select and repeat

```
<book>{
  generate-title-page ($result/meta),
  format-front-matter ($result/front-matter),
  generate-toc ($result/chapter[../meta/toc-style ne “none”]),
  format-chapter ($result/chapter[@type ne “appendix”])
  format-appendix ($result/chapter[@type = “appendix”])
  generate-colophon ($result/meta)
}</book>
```

Calls functions repeatedly, for each selected node

Doesn’t call function if arg is empty sequence

Order of items is preserved

Function mapping is powerful and useful

- Can drive you crazy if you don't realize it's happening

Turn off mapping in the module prolog

```
declare option xdmp:mapping "false";
```

Use XPath final steps if order unimportant

- They're "less non-obvious"

Higher order functions

Pass around functions like other values

```
declare function local:do-something ($arg1, $arg2)
{
   fn:concat ("arg1=", $arg1, ", arg2=", $arg2)
};

let $func := xdmp:function (xs:QName ("local:do-something"))

return xdmp:apply ($func, "Hello", "World")
```

- Strategy Pattern
- Callback
- Dispatch

Strategy Pattern

```
let $search-func := search-algorithm ($user, $request/options)
let $format-func := formatter (($user/theme, $default-theme[1])

return xdmp:apply ($search-func, $user, $request, $format-func)
```

Strategy Pattern

```
let $search-func := search-algorithm ($user, $request/options)
let $format-func := formatter (($user/theme, $default-theme[1])

return xdmp:apply ($search-func, $user, $request, $format-func)
```

```
declare function search-algorithm ($user, $options)
{
  if ($options/sort-order = "des")    (: massively simplified :)
  then xdmp:function (xs:QName ("search-date-des”))
  then xdmp:function (xs:QName ("search-date-asc”))
}

declare function search-date-asc ($user, $request, $formatter)
{ ... };

declare function search-date-des ($user, $request, $formatter)
{ ... };
```

Callback functions

```
declare function search-date-des (
  $user as element(user),
  $request as element(search-request),
  $formatter as xdmp:function
) as element(search-result)
{
  (
    for $doc in cts:search (fn:doc(), ... )
    order by $doc/meta/pub-date descending
    return xdmp:apply ($formatter, $doc, $user)
  )[$request/first to $request/last]
};
```

Dispatch

```
declare variable $search-score-asc := xdmp:function (...)
declare variable $search-score-desc := xdmp:function (...)
declare variable $search-date-asc := xdmp:function (...)
declare variable $search-date-des := xdmp:function (...)

let $sort-key := fn:concat ($req/sort-by, "-", $req/order)

return
  if ($sort-key = "score-asc")
  then xdmp:apply ($search-score-asc, ...)
  else
    if ($sort-key = "score-des")
    then xdmp:apply ($search-score-des, ...)
    else
      if ($sort-key = "date-asc")
      then xdmp:apply ($search-date-asc, ...)
      else ...
```

Dispatch

```
declare variable $search-score-asc := xdmp:function (...)
declare variable $search-score-desc := xdmp:function (...)
declare variable $search-date-asc := xdmp:function (...)
declare variable $search-date-des := xdmp:function (...)

let $sort-key := fn:concat ($req/sort-by, "-", $req/order)

return
  if ($sort-key = "score-asc")
  then xdmp:apply ($search-score-asc, ...)
  else
    if ($sort-key = "score-des")
    then xdmp:apply ($search-score-des, ...)
    else
      if ($sort-key = "date-asc")
      then xdmp:apply ($search-date-asc, ...)
      else ...
```

This is a bit clumsy, better solution shortly.

Check the Map

- Fast caching
- Pre-initialized maps to simplify code
- Dispatch maps

Fast caching with maps

```
declare variable $items :=
   <items>
      <pony id="pinky">Pinky Pie</pony>
      <donkey key="eeyore">Eeyore</donkey>
      <horse id="seabiscuit">Sea Biscuit</horse>
      <aardvarck>Aarnie Aardvarck</aardvarck>   (: no id/key :)
   </items>;
declare variable $id-map := map:map();

(: XPath final step and function mapping in play here :)
$items/*/map:put ($id-map, .//(@id|@key), .),

map:get ($id-map, "eeyore"),
map:get ($id-map, "seabiscuit"),
map:get ($id-map, "pinky")
```

Fast caching with maps

```
declare variable $items :=
   <items>
      <pony id="pinky">Pinky Pie</pony>
      <donkey key="eeyore">Eeyore</donkey>
      <horse id="seabiscuit">Sea Biscuit</horse>
      <aardvarck>Aarnie Aardvarck</aardvarck>   (: no id/key :)
   </items>;
declare variable $id-map := map:map();

(: XPath final step and function mapping in play here :)
$items/*/map:put ($id-map, .//(@id|@key), .),

map:get ($id-map, "eeyore"),
map:get ($id-map, "seabiscuit"),
map:get ($id-map, "pinky"
```

```
<donkey name="eeyore">Eeyore</donkey>
<horse id="seabiscuit">Sea Biscuit</horse>
<pony id="pinky">Pinky Pie</pony>
```

Pre-initialized maps

```
declare variable $country-codes as map:map := map:map (
  <map:map xmlns:map="http://marklogic.com/xdmp/map">
    <map:entry>
      <map:key>us</map:key>
      <map:value>United States</map:value>
    </map:entry>
    <map:entry>
      <map:key>uk</map:key>
      <map:value>United Kingdom</map:value>
    </map:entry>
    ...
  </map:map>
);

...

process ($request, map:get ($country-codes, $request/@ccode))
```

Pre-initialized function dispatch maps

```
declare variable $search-funcs as map:map := map:map (
  <map:map xmlns:map="http://marklogic.com/xdmp/map">
    <map:entry>
      <map:key>score-asc</map:key>
      <map:value>xdmp:function (xs:QName ("scorea"))</map:value>
    </map:entry>
    <map:entry>
      <map:key>score-des</map:key>
      <map:value>xdmp:function (xs:QName ("scored"))</map:value>
    </map:entry>
    ...
  </map:map>
);

let $sort-key := fn:concat ($req/sort-by, "-", $req/order)

return
xdmp:apply (map:get ($search-funcs, $sort-key), $request, ...)
```

Potpourri

Keep it to yourself

Be lazy whenever possible

Just don’t touch anything

You touch it, you own it

Keep it to yourself

- Use private variables and functions where possible
- Prevents seeing inside a module's implementation

```
declare private variable $paras-to-ignore :=
  ( "prepub", "beta", "withdrawn" );

declare private function extract-paras ($doc) {
  $doc//para[fn:not (@state = $paras-to-ignore)]
};

declare private function xform-para ($paras as element(para))
{ ... };

declare function process-paras ($doc as element(article))
  as element(para)*
{
  xform-para (extract-paras ($doc))   (: function mapping :)
}
```

Keep it to yourself

- Use private variables and functions where possible
- Prevents seeing inside a module's implementation

```
declare private variable $paras-to-ignore :=
  ( "prepub", "beta", "withdrawn" );

declare private function extract-paras ($doc) {
  $doc//para[fn:not (@state = $paras-to-ignore)]
};

declare private function xform-para ($paras as element(para))
{ ... };

declare function process-paras ($doc as element(article))
  as element(para)*
{
  xform-para (extract-paras ($doc))   (: function mapping :)
}
```

```
Bonus trick:
inverting an
existential test
<==
```

Be lazy whenever possible

Module variables are initialized lazily

Never initialized if not referenced

```
declare variable $complicated := expensive-query ($foo, $bar);
declare variable $simple := cheap-query ($foo, $bar);

if ($request/include-everything)
then process ($complicated)
else process ($simple, $default-values)

(: only one of expensive-query() or cheap-query will be run :)
```

Just don’t touch anything

- Making sure you run as a query, not an update
- Updates lock every document they touch
- Sometimes you want to look at many, many documents, but only update a few - or none.
  - Any updates will be done via xdmp:eval/invoke
- Insure the main request is always a lock-free query

```
declare variable $enforce-query :=
  if (xdmp:request-timestamp())
  then ()
  else fn:error((), "Must run as query");

$enforce-query,          (: remember lazy init of global vars :)
do-something()
```

You touch it, you own it

- Some builtins can run concurrently (ie, lexicons)
- They return immediately and work asynchronously until complete
  - If you reference the result, you block until it's ready

Maximize concurrency by waiting as long as possible

```
(: cts:element-values can run concurrently, do them first :)

let $publishers := cts:element-values (xs:QName("publisher"))
let $pub-types := cts:element-values (xs:QName("pub-type"))
let $black-balled := get-black-balled-publishers (...)
let $featured := get-featured-items (...)

return
process ($publishers, $pub-types, $featured, $black-balled)
```

What about xdmp:set()?

- Modifies the value of a variable

I'm not a fan

- Breaks the functional programming model
  - Can defeat parallelism as XQuery implementation improves
- Too easily used as a hack to make XQuery "easier"
- There's nearly always a way to avoid using it

Same for maps

- I'm guilty - see earlier slide re: caching with maps
- I'd like to see an immutable map type

Send Me Your Own Tricks

If you know any clever / stupid XQuery tricks, please send them along to me:

ron@overstory.co.uk

Slides at:

http://github.com/overstory

Questions?

Ron Hitchens
Principal Consultant, OverStory Ltd
ron@overstory.co.uk
http://github.com/overstory