MarkLogic™

USER CONFERENCE 2011

SAN FRANCISCO | APRIL 26 – 29

Query Tuning Basics

DEVELOPER LOUNGE LAB

Table of Contents

Step 1: Install MarkLogic Server

Download from here: http://developer.marklogic.com/products

Step 2: Get the data

a. Download from here: http://developer.marklogic.com/media/mluc11-labs/performance-tuning/all_xml.zip (Original data source: http://www.weather.gov/xml/current_obs/)
b. Unzip to a folder on your hard drive, e.g., /Users/elenz/Downloads/all_xml

Step 3: Load the data

a. Open the MarkLogic Server home page - http://localhost:8000
    i. Click "Load new content"
b. Under "Collect" click the "Configure" button
c. Type the directory path of where you unzipped the XML on your machine, e.g., /Users/elenz/Downloads/all_xml
d. Click "Done"
e. Under "Load" click the "Document settings" button
f. Edit the URI field so it has this value: **/weatherData/{$filename}{$dot-ext}**
g. Click "Done"
h. Leave the database as "Documents"
i. Click "Start Loading"
j. Wait until progress bar indicates that loading is complete

Step 4: Install CQ

a. Copy the "Samples/cq" directory into the "Docs" directory inside the directory for your MarkLogic installation, e.g., on Mac OS X, copy ~/Library/MarkLogic/Samples/cq into ~/Library/MarkLogic/Docs.
b. Test CQ by opening http://localhost:8000/cq in your browser

Step 5: Learn how to write fast queries!

Any time a query is listed below, type it in CQ and hit the "Text" button to show the results. Make sure you have the same database (e.g., "Documents") selected that you loaded the content into.

Let's look up one document in the database by its URI:

```
(:1:)   doc("/weatherData/KDMO.xml")
```

This query runs fast because it uses MarkLogic's Universal Index to look up the document based on one of the many facts that are indexed about documents (in this case, its URI). (By the way, what *is* the Fahrenheit temperature at Sedalia Memorial Airport anyway? Hint: see <temp_f> in the document we just retrieved. _____)

Here's an equivalent way to run the same fast query. This time let's tack on xdmp:query-meters() to the end to see why it's fast. (xdmp:query-meters() returns an XML document describing various performance characteristics of the query run just before it.)

```
(:2:)   collection()[base-uri(.) eq "/weatherData/KDMO.xml"],
        xdmp:query-meters()
```

In the results pane, scroll down past the document we retrieved to the <qm:query-meters> element (the output of xdmp:query-meters). This has a lot of information, but let's focus on just three parts:

How long did the query take to run (its elapsed time)? _____
How many expanded tree cache hits were there? _____
How many expanded tree cache misses were there? _____

An "expanded tree" is an un-compressed XML document (actually, it's technically an un-compressed XML *fragment*, but since we haven't enabled fragmenting in this tutorial, we can assume that a document is the same as a fragment). A "cache hit" means an expanded document was cached and read straight from memory. A "cache miss" means the document had to be grabbed from either the "compressed tree cache," or, if not cached there, then read off the disk. Either way, the above query was fast, because it only required accessing (reading) a total of 1 document.

Now, let's run an intentionally slow query:

```
(:3:)   collection()[starts-with(base-uri(.), "/weatherData/KDMO")],
        xdmp:query-meters()
```

This query does not make effective use of MarkLogic's indexes. It is likely to run slowly the first time you run it.

After running it once:

How long did the query take to run (its elapsed time)? _____
How many expanded tree cache hits were there? _____
How many expanded tree cache misses were there? _____

Since MarkLogic also caches the documents it reads into memory, it will run much faster on subsequent runs. After running the above query again:

How long did the query take to run (its elapsed time)? _____
How many expanded tree cache hits were there? _____
How many expanded tree cache misses were there? _____

For performance with a small data set (like the one we're using in this tutorial), relying on the in-memory cache may suffice. But relying solely on caching is not a scalable approach. We're only retrieving (what turns out to be) one document, yet the query is reading every document in the database. We need a faster way to access the values that are present without having to open each and every document. Whereas the Universal Index (the default, always-on index) provides fast *document* lookups given a specific value, we need a way to get fast *value* lookups. That's what *lexicons* (another kind of index) are for. In this case, the lexicon we want to enable is the "URI lexicon." Before we do that, let's try to run the query that accesses the URI lexicon:

```
(:4:)   cts:uris()
```

If you do not already have the URI lexicon enabled, this will produce the error: "URI lexicon not enabled." In another browser window or tab, go to http://localhost:8001, choose "Databases", click "Documents", and scroll down to the "uri lexicon" form field. Select "true" where it says "Maintain a lexicon of document URIs." Scroll to the bottom and hit the "ok" button. This will cause MarkLogic to immediately start reindexing your content so that cts:uris() will work. Click the "Status" tab to check on the status of indexing. Hit refresh until you see that "Reindexing/Refragmenting State" says "Not reindexing/refragmenting."

Now run the above query again. This will give you a sorted list of all the document URIs in your database. Now let's write a faster version of our original query:

```
(:5:)   for $uri in cts:uris()[starts-with(., "/weatherData/KDMO")]
        return doc($uri),
        xdmp:query-meters()
```

Now:

How long was the query's elapsed time? _____

How many expanded tree cache hits were there? _____
How many expanded tree cache misses were there? _____

The goal throughout the rest of this tutorial will be to find ways to maximize our use of indexes (whether default or manually enabled) so as to *prevent excess document reads*. That is the single most important factor in writing fast, scalable queries against MarkLogic Server. In general, query evaluation in MarkLogic has two stages:

1. Through the use of *index resolution*, narrow down the set of possible documents that might be relevant, and
2. Read all the documents in the narrowed set, *filtering* out which ones actually aren't relevant.

In other words, we want to prevent excess filtering. To be sure, filtering is a good thing; it's what ensures that our final results are accurate. But index resolution is what makes queries *fast* (and scalable).

Run the following query:

```
(:6:)   count(/current_observation)
```

NOTE: MarkLogic provides this short cut in its XQuery implementation: If you type "/" at the beginning of a top-level path expression, it is interpreted to be short for "collection()/"; we'll continue to utilize this shortcut when convenient in the rest of this tutorial.

How many <current_observation> docs are in the database? _____

Now run the following query. Whereas count() is guaranteed to be accurate (because it performs filtering), xdmp:estimate() is guaranteed to be fast (because it skips filtering):

```
(:7:)   xdmp:estimate(/current_observation)
```

How many <current_observation> docs are there estimated to be (from index resolution alone)? ____

Some of the documents in our data set contain a <weather> element, and some do not. Run this query to find out how many do:

```
(:8:)   count(/current_observation[weather])
```

Result: _____

And run this query to see how accurate index resolution was:

```
(:9:)   xdmp:estimate(/current_observation[weather])
```

Result: _____

As you can see, in certain situations, xdmp:estimate() can be accurate (returning the same value as count()). You will need to determine yourself whether it makes sense to use xdmp:estimate() as an optimization even in cases where accuracy is important; a deep understanding of your own data set and how MarkLogic indexes your data will be necessary for this. Otherwise, using count() is the safest bet for when accuracy is important. (In most cases, xdmp:estimate() is used indeed as an *estimate*, in the same way that Google estimates how many search results your query returned.)

Conversely, let's find out how many documents do *not* have a <weather> element:

```
(:10:)  count(/current_observation[not(weather)])
```

Result: _____

By comparison, how accurate is index resolution for this query?

```
(:11:)  xdmp:estimate(/current_observation[not(weather)])
```

Result: _____

Here we see a case where the index was *not* able to narrow down the set of documents (beyond those that have <current_observation> as their root element). That's because the fact that a document does *not* have a particular element is not something that is included in the index. Instead, the index contains facts ("terms") that can be automatically determined from the document itself, such as what words it contains, what elements it contains, what parent/child relationships it has, and what ancestor/descendant relationships it has. In general, the index can only look up documents based on the *presence* of a word/element/etc., not the absence of one.

Let's take a peek at how MarkLogic translates XPath expressions into queries against its indexes. Run the following query:

```
(:12:)  xdmp:plan(collection()/current_observation[weather]),
        xdmp:plan(collection()/current_observation[not(weather)])
```

xdmp:plan() outputs an XML document representing the execution plan for the given expression. It, like xdmp:estimate(), is a pseudo-function (or "special form" a la Lisp) and has the same "signature" as xdmp:estimate(). You must pass it either a searchable XPath expression or a call to cts:search() (more on that later). Like xdmp:estimate(), it runs the index resolution stage but does not perform the filtering stage. The above query returns two <qry:query-plan> documents.

In the first query plan, did the “Step 2 predicate” contribute any "constraints"? _____
How many fragments (documents) are selected for filtering (see <qry:result>)? _____

In the second query plan, did the “Step 2 predicate” contribute any "constraints"? _____
How many fragments (documents) are selected for filtering (see <qry:result>)? _____

As you can see, testing for the presence of <weather> allows MarkLogic to make greater use of its indexes than does testing for its absence.

This indexing model is used not only to make XPath fast; it's also used in (and originally comes from) the world of full-text search. XPath itself does not have full-text search semantics; MarkLogic provides extension functions for adding that support.

We'll continue to examine how many fragments (documents) each query selects via index resolution. Besides xdmp:plan(), another convenient way to see index resolution in action is to use xdmp:query-trace(). Rather than returning its information, it outputs its information to the error log. It prints out much of the same info that is present in the result of xdmp:plan() (specifically, everything inside the <qry:info-trace> elements). This is convenient, because you can just add it to the top of your query; your expressions will then be evaluated as usual while the query trace info is printed to the error log.

Run the following query:

```
(:13:)  xdmp:query-trace(true()),

        let $results := /current_observation[contains(location,'Airport')]
        return count($results)
```

What was the result of this query (in the result pane)? _____
How many fragments had to be filtered (see the error log)? _____

Despite only a subset of the documents containing the string "Airport", all of the documents had to be filtered. Why didn't the index resolution help us out more? The answer is that MarkLogic does not index every possible substring that appears in a document (that would balloon the data size to a ridiculous degree). Instead, it indexes words (tokens separated by space or punctuation), plus word stems and case-insensitive (lower-case) and case-sensitive versions of the words as they appear. To query that information, we need something other than the XPath contains() function.

Run the following query (**from now on, I'll assume you're adding "xdmp:query-trace(true())," at the top of your CQ query field**):

```
(:14:)  let $results := /current_observation[cts:contains(location,'Airport')]
        return count($results)
```

What was the result of this query (in the result pane)? _____
How many fragments had to be filtered (see the error log)? _____

As you can see, the two queries return the same results, but the second one (using cts:contains()) makes more effective use of the indexes. Keep in mind that the contains() and cts:contains() functions do not have the same meaning; they just happen to return the same results in this case. To drive this point home, let's try a variation of the above two queries (notice the lower-case "airport"):

```
(:15:)  let $results := /current_observation[contains(location,'airport')]
        return count($results)
```

What was the result of this query (in the result pane)? _____
How many fragments had to be filtered (see the error log)? _____

And now for the cts version:

```
(:16:)  let $results := /current_observation[cts:contains(location,'airport')]
        return count($results)
```

What was the result of this query (in the result pane)? _____
How many fragments had to be filtered (see the error log)? _____

contains() is always case-sensitive and tests for the presence of any substring. cts:contains(), on the other hand, uses search-style semantics; it may be either case-sensitive or case-insensitive. The default when passing it an all-lower-case string is to do a case-*insensitive* search. Run this query to see a specific example of how contains() means something different than cts:contains():

```
(:17:)  /current_observation[contains(location,'airport')]
```

As you can see, "Fairport" returns true for contains(location,'airport') but not for cts:contains(location,'airport'), since "airport" is not a word in this case.

It's been an unseasonably cold spring where I live, so let's find some warm places. Run this query:

```
(:18:)  let $results := /current_observation[temp_f eq '85.0']
        return count($results)
```

What was the result of this query (in the result pane)? _____
How many fragments had to be filtered (see the error log)? _____

That gives me a set of places that have measured exactly 85 degrees Fahrenheit, which is a nice start. And index resolution made the query fast too, because I provided a specific value ("85.0"), which MarkLogic stores in its Universal Index. But really, I'd be happy to visit places where the temperature is, say, 86 degrees, or 85.5 degrees, or even anything higher than 85 degrees. So let's try a different query:

```
(:19:)  let $results := /current_observation[number(temp_f) ge 85.0]
        return count($results)
```

What was the result of this query (in the result pane)? _____
How many fragments had to be filtered (see the error log)? _____

Okay, now that gives me some more places to choose from. But now the query won't scale well, because it's not making good use of indexes (it's having to filter too many fragments). In this case, I'm no longer providing a specific value. Instead, I'm looking for a *range* of values. As with the cts:uris() example, if I want to be able to do fast *value* lookups, I'll need a lexicon for this—in this case, an element value lexicon, which MarkLogic calls an "element range index." Whether I want to look up an element's distinct values or query them for a range of values, MarkLogic uses the same mechanism: element range indexes.

First, let's see what happens if we try to retrieve all the distinct values of <temp_f> that are present in the database. First, using regular XQuery/XPath:

```
(:20:)  let $ordered-values :=
        for $v in distinct-values(/current_observation/temp_f) order by
        number($v) return $v
        return
        (
          concat("Total count of values: ", count($ordered-values)),
          $ordered-values
        )
```

What is "Total count of values"? _____
What's the lowest temperature reading? _____
What's the highest temperature reading? _____
What value appears that doesn't "belong" (hint: it's *not applicable*)? _____
How many fragments had to be filtered (see the error log)? _____

Okay, so that was the slow way. Now let's try the fast way:

```
(:21:)  let $ordered-values :=
        cts:element-values(xs:QName('temp_f'))
        return
        (
          concat("Total count of values: ", count($ordered-values)),
          $ordered-values
        )
```

Unless you're one step ahead and have already created the range index, you'll get an error: "No element range index for fn:QName("", "temp_f"). In another browser window or tab, go to http://localhost:8001, choose "Databases", click "Documents", and then choose the "Element Range Indexes" link in the left-hand navigation. Click the "Add" tab to add a new range index. Set scalar to "decimal", leave namespace URI blank, set localname to "temp_f", and leave range value positions at "false". Then click the "ok" button. Scroll to the bottom and hit the "ok" button. This will cause MarkLogic to immediately start reindexing your content so that cts:element-values() will work. Go back to the database configuration page and click the "Status" tab to check on the status of indexing. Hit refresh until you see that "Reindexing/Refragmenting State" says "Not reindexing/refragmenting."

Now try running the above query again.

What is "Total count of values"? _____
What's the lowest temperature reading? _____
What's the highest temperature reading? _____
Does "NA" appear anymore? _____
How many fragments had to be filtered (hint: the error log won't help you here)? _____

Now, the range index values (which are all stored in memory) are immediately accessible without reading from any fragments at all. (xdmp:query-trace() doesn't reveal anything in the error log for this query because it contains no path expressions or cts:search() calls.) If you're wondering why there's a discrepancy between the "Total count of values" in the above two queries, the answer is that the indexer coerced each number into a decimal as we instructed, whereas our first (slow) query did not. Thus "40" and "40.0" are treated as distinct values in the first query but as the same value in

the second. Note also that "NA" does not appear in the list here, since it could not be coerced by the indexer into a decimal value.

Now we're finally ready to run our range query. To express a range query that utilizes our range index, we could just use a regular XPath predicate, but we can also use a “cts query”, which in this case will allow us to squeeze even more performance out of our index. We’ll construct a special kind of value called a "cts:query" and then pass it to cts:search() (another special form) to be evaluated. The first "argument" to cts:search() is a searchable XPath expression. The second is a cts:query value. MarkLogic Server provides a number of different cts:query constructor functions, which can be combined in various ways. In the following query, we're telling the server to hold off on fully executing the given call to cts:search() and instead just run the index resolution stage, skip the filtering stage, and yield the number of fragments (documents) found via the index:

```
(:22:)  xdmp:estimate(
          cts:search(/current_observation,
            cts:element-range-query(xs:QName("temp_f"), ">=", 85.0)))
```

What's the result of this query? _____
What was the result of the earlier query (#19) that used regular XPath? _____

This yields a lot more potential results than the XPath expression did. Any idea why? As a hint, let's see what the following expression returns:

```
(:23:)  count(/current_observation[temp_f eq 'NA'])
```

What's the result of this query? _____

Do you see a relationship between the last three numbers you wrote down? (Hint: A - B = C) This suggests that we need the filtering stage to filter out all those "NA" values (weather observations in which a temperature reading is apparently not available). Let's try running the full search now, including the filtering stage:

```
(:24:)  let $results :=
          cts:search(/current_observation,
            cts:element-range-query(xs:QName("temp_f"), ">=", 85.0))
        return
        count($results)
```

You should see this error message: Invalid cast: xs:untypedAtomic("NA") cast as xs:decimal.
Just as the indexer could not cast "NA" to a decimal, neither can the query do so when we force it to evaluate it all the way through the filtering stage. We can fix this by explicitly filtering out those documents that have "NA" as their <temp_f> value:

```
(:25:)  let $results :=
          cts:search(/current_observation[not(temp_f eq "NA")],
            cts:element-range-query(xs:QName("temp_f"), ">=", 85.0))
        return
        count($results)
```

What's the result of this query? _____
How many fragments had to be filtered (see the error log)? _____

This is a step in the right direction. In fact, it may be fast enough. If we know that there will be only so many appearances of "NA" in the database, then we won't have a scalability problem. But we will have a problem if more and more instances of "NA" show up. In that case, we can tune the query even further by weeding out the "NA" values at the index resolution stage:

```
(:26:)  let $results :=
          cts:search(/current_observation,
            cts:and-not-query(
```

```
            cts:element-range-query(xs:QName("temp_f"), ">=", 85.0),
            cts:element-value-query(xs:QName("temp_f"), "NA")
          )
        )
      return
      count($results)
```

What's the result of this query? _____
How many fragments had to be filtered (see the error log)? _____

Now we are getting accurate results from index resolution alone and could choose to use xdmp:estimate() if it's only the count that we're after.

Bringing us full circle, we now have a hint as to how we might get accurate results from the index as to how many documents do *not* contain a <weather> element. We can first find all documents that *do* have a <weather> element and subtract that set (using cts:not-query()) from the entire set of <current_observation> documents:

```
(:27:)  xdmp:estimate(
          cts:search(/current_observation,
            cts:not-query(
              cts:element-query(xs:QName("weather"), cts:and-query(()))))
```

What's the result of this query? _____
Are the results accurate? (Does it produce the same result as query #10?) _____

cts:not-query() returns a query that matches all document IDs that do *not* match the given cts:query argument. cts:element-query matches all documents containing an element that has the given QName and that matches the given cts:query in its second argument. cts:and-query() with an empty sequence yields a query that always matches.

You should take great care with cts:not-query(). Once you start using it, you have the potential of getting false negatives (in other words, missing results). It is much safer to use positive queries and then allow the filtering stage to weed out any false positives. If you decide to use cts:not-query (or cts:and-not-query as above), you'll need to make sure that the query you pass to it does not yield any false positives, because they will potentially translate to false negatives; even if filtering is enabled, you won't have a chance to get them back. See http://developer.marklogic.com/pubs/4.2/apidocs/cts-query.html#cts:not-query

That concludes our tour. Well done! You now know way more about how to write fast queries than the average XQuery developer does. Next quest: Try out the "Profile" button in CQ. Also, check out prof:invoke(), etc.: http://developer.marklogic.com/pubs/4.2/apidocs/ProfileBuiltins.html

Step 6: Dig deeper

You can find additional information at the following locations:

- Query Performance and Tuning Guide
  - http://developer.marklogic.com/pubs/4.2/books/performance.pdf
- Search Developer's Guide
  - http://developer.marklogic.com/learn/4.2/search-dev-guide
- "How Indexing makes XPath Fast"
  - http://developer.marklogic.com/blog/how-indexing-makes-xpath-fast
- "Inside MarkLogic Server"
  - http://developer.marklogic.com/inside-marklogic

If you have any questions or comments, post them as a comment on this page: http://developer.marklogic.com/blog/how-to-write-fast-queries

- Evan