Fast Faceted Search in XML

Using XQuery and Indexes in eXist-db

Anne Schuth
anne.schuth@uva.nl

Maarten Marx
maartenmarx@uva.nl

Abstract

We present and compare three implementations of faceted search in eXist-db, an XML database. The bit-vector based implementation outperforms the other two implementations in that performance is near constant when the size of the database grows. We investigate this method in detail to pinpoint the source of this speedup. We do so using a micro-benchmark based on XMark designed for evaluating faceted search.

1 Introduction

For a long time, search engines were geared toward expert users that are aware of the content of the database being searched. However, the average user of a web search interface has no clue about this content and often even less about the exact phrasing of what is being searched. It has been shown useful to provide users with suggestions (Hearst et al., 2002), and even more useful if a user knows how many results a refinement of his query would solicit *before* the actual refinement takes place. This concept is commonly called *faceted search* (Burke et al., 1996; English et al., 2002; Hearst, 2006, 2008, 2009) and is not specific to any form of data storage. A database of XML documents seems very suitable to faceted search; XML by nature is fit for storing metadata in the form of arguments or meta elements. To illustrate, see Figure 1 (following page) for screen-shots of two real world examples of faceted search.

Facets consist of meta data; in the case of a database with cars, natural candidates for facets are *'color'*, *'brand'*, *'price'*, etc. Each of these facets can take a *value* (a nominal value: *'red'*, *'blue'*, etc. for the *'color'* facet and a range *'0-1000'*, *'1000-2000'*, etc for the *'price'* facet). Then, when a user starts searching by typing for example *'4 wheel drive'*, the respective amounts of retrieved *'red'* and *'blue'* 4 wheel drive cars are displayed along with all results for that query and the user can *drill-down* with a simple mouse click on facet-value *'red'* or *'blue'*. For the user, this has the benefit that much of the content of the database is exposed, both in vocabulary and in volume. The gap between searching and browsing is bridged (Sacco and Tzitzikas, 2009).

The calculation of the counts for these *facet-values* is the problem this paper addresses. Simple pre-calculation of these counts is infeasible; the users' freedom of using free-text queries forbids this. The technical challenge is the efficient partitioning of the results of a (usually free-text) query and computing the cardinality of each block. How a result set is partitioned depends on the data, the application, and in some cases also on the result set itself (Dash et al., 2008).

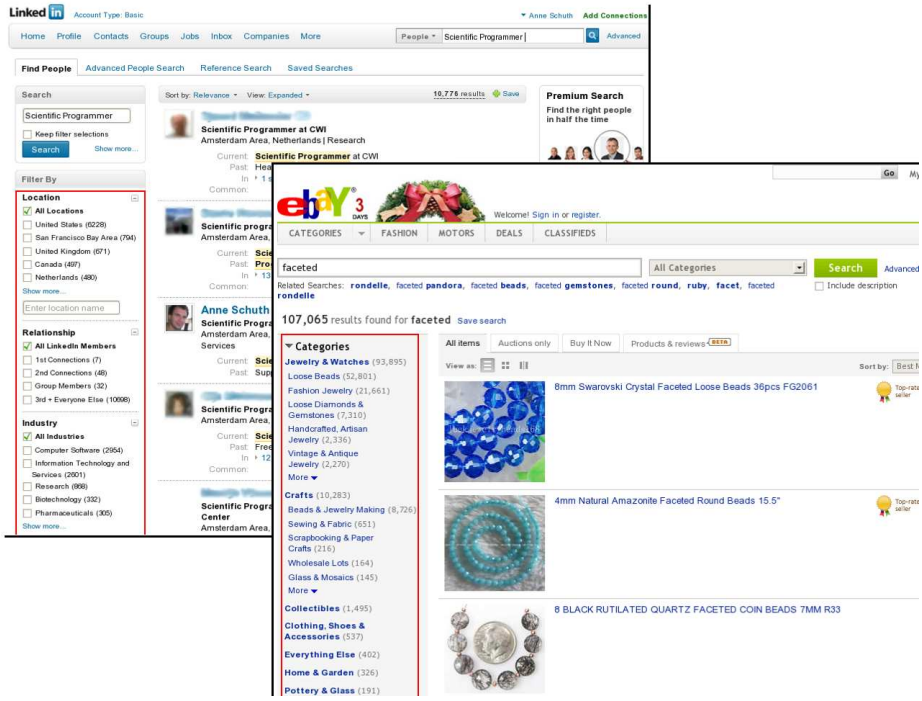

Figure 1: Screen-shots of examples of faceted search at LinkedIn and eBay, after searching with free-text query 'scientific programmer' and respectively 'faceted'. The facets, their values and counts are in the red boxes.

The contribution of this work is four-fold. We specify an expressive XQuery language that can capture all aspects of faceted search. We compare 3 implementations on performance. We introduce an easy to deploy micro-benchmark geared toward faceted search. And finally we contribute a faceted search extension to the eXist-db database.

Our experiments show that for smaller datasizes all methods perform fine for both of our benchmark scenarios. However, the time taken by both the *Naive Method* and the *Rvdb Method* grows with the datasize while the time taken by the *Indexed Method* stays near constant. We can conclude that the bit-vector implementation is the only one which scales. Our average running time of around 1 second for all queries on all datasets (going up to almost 300K nodes or 1.5GB) is promising but not good enough. There are two obvious starting points for improvements: find the reason for the outlier in our experiment, see Figure 4 (p. 12) and 5 (p. 13), and implement a solution like Wang (2009) for fast mappings to and from positions in a bit-vector to node-identifiers.

This paper is organized as follows. In the next section we describe and define our XQuery language features. We then introduce two benchmark scenarios that we use to compare the three different implementations that are presented in section 4 (p. 7). We describe and present the results of our experiments in section 5 (p. 10), after which we describe related work and draw conclusions.

2 Specification Language

2.1 XQuery

When dealing with a database of XML documents, XQuery (Boag et al., 2007) is the language of choice. XQuery is more than just a query language, it is a functional programming language aimed at transforming XML into XML. Our database, eXist-db, supports XQuery as a language to develop web-applications making it even more fit for implementing faceted search, which is an interface related subject. We extend the XQuery language with 3 functions, described in the following sections. We define them in the namespace —a module— `facet`.

2.1.1 Counts

The main goal of our work is implementing an efficient `counts()` function, as this is where the heavy computation takes place. This function should take any arbitrary sequence and return an XML node that holds counts for all facet-values that are defined on the items in that sequence. We define the structure of this node using RelaxNG (Clark et al., 2001) as follows:

```
start = element facets { Facet* }
Facet = element facet {
          attribute name { xsd:NCName }, Value*
        }
Value = element value {
          attribute name { text },
          xsd:integer
        }
```

An example would look like:

```
<facets>
  <facet name="location">
    <value name="United States">1614</value>
    <value name="Barbados">8</value>
    <value name="Cape Verde">6</value>
    <value name="Samoa">5</value>
    <value name="Cacos Islands">4</value>
  </facet>
  <facet name="quantity">
    <value name="1">2006</value>
    <value name="2">162</value>
    <value name="3">6</value>
    <value name="4">1</value>
  </facet>
</facets>
```

The signature of the `counts()` function that returns this data-structure for a sequence of nodes is specified as follows:

- `facet:counts($nodes as node()*) as node()`
  Typically `$nodes` is a sequence representing hits from, for example, a full-text query. The returned node complies with the schema above.

A common usage would be firing a full-text query (for example '*gold*') and then requesting the facet-value counts to display them along with the results:

```
let $hits := $doc//item[facet:query(., 'gold')]
return
  <result>
    <hits>{$hits}</hits>
    {facet:counts($hits)}
  </result>
```

2.1.2 Filter

In a drill-down scenario, always a certain facet-value is chosen. We need a way to filter a node sequence with this facet-value and therefore introduce the following function:

- `facet:filter($nodes as node()*, (($facet as xs:string, $value as xs:string)*)) as node()*`
  Where a sequence of `($facet, $value)` pairs, in the form of strings, should be specified and `$nodes` is the sequence of nodes that has to be filtered. The filtered sequence is returned.

Note that this function can be used in conjunction with a full-text query, for example like:

```
let $hits := $doc//item[facet:query(., 'gold')][facet:filter(., ('location', 'Samoa'))]
return
  <result>
    <hits>{$hits}</hits>
    {facet:counts($hits)}
  </result>
```

Note that it would be easier to optimize a single call to the `filter()` function than multiple calls in a row.

In other words, the statement

```
$doc//item[facet:filter(., (('location', 'Samoa'), ('quantity', '2')))]
```

is probably implemented more efficiently than

```
$doc//item[facet:filter(., ('location', 'Samoa'))][facet:filter(., ('quantity', '2'))]
```

2.1.3 Add

Apart from the method described in section 2.3 (next page), it is convenient to be able to programmatically add —store— the bit-vector for a facet-value on the fly. Such a feature would be useful to cache queries that are often used or to allow a user to store a query for later reuse. The add function is defined as follows:

- `facet:add($nodes as node()*, $facet as xs:string, $value as xs:string) as xs:boolean`
  Where `$nodes` is the sequence that should be stored in the facet `$facet` under value `$value`. True is returned on success.

Note that using this function, arbitrary sequences can be stored as a facet-value, potentially making little sense to an end user. Also note that this function is not a function in the mathematical sense, it has a side effect.

```
let $hits := $doc//item[facet:query(., 'gold')][facet:filter(., ('location', 'Samoa'))]
return facet:add($hits, 'user-queries', 'gold-samoa')
```

2.2 XPath

The functions we use in XQuery are effectively also functions in XPath (Berglund et al., 2007), which is a subset of the XQuery language. So we get additional XPath functionality for free. This is an advantage since it means we can use the same functions in XSLT as well.

2.3 Index Specification

For our indexed method only, see Section 4.3 (p. 9), there needs to be some way of telling the indexing engine what nodes we are considering to be 'documents', what the facets are and where in the database their values can be found. The eXist-db database has a mechanism in the form of a `collection.xconf` file for each collection, to configure its indexes. We extended the syntax for this file to allow for specifying facets. The syntax for the index definition of facets is defined as follows and replaces the `lucene` element, of which the syntax is partially borrowed:

```
Facets = element facets { Text* }
Text = element text {
         attribute qname { xsd:NCName },
         Facet*
       }
Facet = element facet {
          attribute name { xsd:NCName },
          attribute select { text { pattern = "\.?(/|//)?\@?\w+" } },
          attribute type { "path" | "multi" | "simple" }?
        }
```

As an example, the full `collection.xconf` definition we used for our benchmark, see the next section, is this:[1]

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <index>
    <fulltext default="none" attributes="no"/>
    <facets>
      <text qname="item">
        <facet name="location" select="location"/>
        <facet name="quantity" select="quantity"/>
        <facet name="featured" select="./@featured"/>
        <facet name="category" select="incategory/@category" type="multi" />
      </text>
    </facets>
    <create path="//location" type="xs:string"/>
    <create path="//quantity" type="xs:string"/>
    <create path="//@featured" type="xs:string"/>
  </index>
</collection>
```

This example tells the engine that for any `item` element it encounters while indexing, it should look for a `location` and `quantity` element under it and for a `feature` and `category` attribute at the designated locations.

The three `create` elements tell the engine that it should also create range indexes for any `location`, `quantity` and `@featured`. These indexes will be used by both the *naive* and *rvdb* method, see Section 4.1 (p. 7) and 4.2 (p. 8).

3 Benchmark

To test in how far one approach outperforms another, we developed a task specific benchmark, a so-called micro-benchmark (Afanasiev et al., 2005, 2006). Such a benchmark is aimed at evaluating very specific features of a language or an optimization. In our case we have three different implementations for the same semantics and we want to compare their performance. None of our three implementations is better in that it adds functionality to our language; each implements the function `counts()`, as in Section 2.1.1 (p. 3).

[1] We use *path* attributes instead of *qname* attributes in the range index definitions because of a known bug in the eXist-db implementation of the *util:index-keys* function, which has been fixed in later releases.

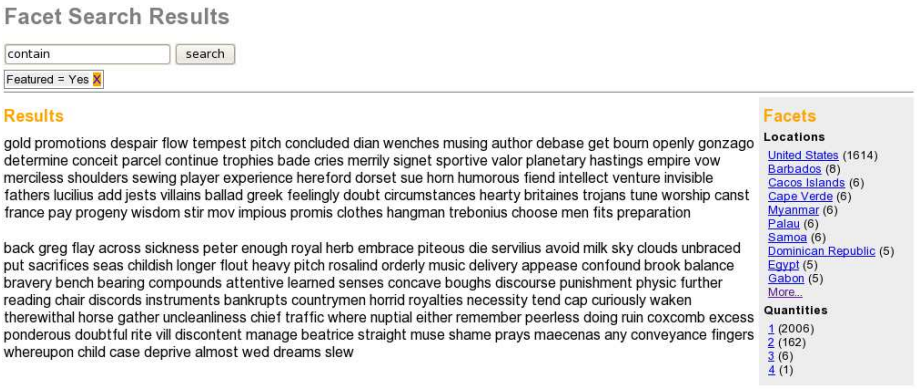

Figure 2: A screen-shot of a faceted search interface on top of the XMark database.

XMark scaling	Size (MB)	# Files	# items
0.1	12	69	2175
0.2	24	136	4350
0.4	46	272	8700
0.8	92	542	17400
1.6	184	1084	34800
3.2	368	2166	69600
6.4	735	4330	139200
12.8	1500	8660	278400

Table 1: The data sizes, in megabytes and number of items, for different scaling factors.

The scenarios use the XMark dataset (Schmidt et al., 2002) in various sizes, with scaling factor 0.1 up to 12.8, see Table 1 for details.[2] The XMark database models an Internet auction site —much like eBay— and is therefore well suited for our use-case. In our scenarios we envision an item search engine, where *items* are the objects that are for sale; the *item* nodes are the 'documents'. A screen-shot is depicted in Figure 2. We use three properties of these items as their facets: *location* (up to 232 values), *quantity* (up to 7 values) and *featured* (two values). A full-text search index is defined on all text —*//text()*— below the *item* node, this index can be queried using the free text input box.

3.1 Scenario A

A first scenario does not use the full-text index. The user arrives at an interface — a website— and is presented with *all* results (although only the top n is displayed) together with the facet-values and their counts. He then consecutively drills down by clicking facet-value 'location' = 'United States', 'quantity' = '2' and 'featured' = 'yes'. The relative data sizes for these four steps are given in Table 2.

[2]The exact command we used is ./xmlgen.Linux -f [factor] -s 100

scenario	step	full-text	facets 'location'	facets 'quantity'	facets 'featured'	% items
A	1	-	-	-	-	100%
	2	-	'United States'	-	-	76%
	3	-	'United States'	'2'	-	5%
	4	-	'United States'	'2'	'yes'	0.5%
B	1	'gold'	-	-	-	6.7%
	2	'gold'	-	'1'	-	6.2%
	3	'gold'	'Barbados'	'1'	-	0.05%
	4	'gold'	'Barbados'	'1'	'yes'	0%

Table 2: Approximate relative data sizes in percentages of the total number of items for each step in Scenarios A and B. Also see Table 1 (page before), for absolute size per dataset.

In this scenario, the first two steps are not very restrictive, so each method has to deal with large portions of the data.

3.2 Scenario B

In the second scenario, Scenario B, we start off with *Q14* from the original XMark proposal (Schmidt et al., 2002). Again, see Table 1 (preceding page) for sizes of the dataset. This scenario is very different from Scenario A in that it starts off with a much smaller portion of the datasets. Also, the last step is a bit artificial: it does not return any results for any of the datasets (a user would probably never end up in that situation). It is still interesting, however, to analyze how efficient each method is in that border case.

4 Three Methods

After Van den Branden (2010), we test 2 methods and call them the *Naive Method* and the *Rvdb Method*. Additionally, we introduce an index and bit-vector based method which we call the *Indexed Method*. Only the *Naive Method* is pure XQuery, the others use eXist-db specific features that exploit its indexes. Each of these three methods implements the function `counts()`, see 2.1.1 (p. 3), that takes a sequence of hits and returns counts for facet-values.

4.1 Naive Method

A simple and naive way of computing facet-value counts would be firing the free-text query together with all facet-values and record the lenght of each returned sequence. For a small dataset and a small number of facet-values this might work, but the cost of this method grows linearly with the number of facet-values, potentially harming performance severely. Our *Naive Method* does exactly this. In Listing 1 (following page) we show the implementation of the *Naive Method* by Van den Branden (2010), as we used it in our experiments.

It should be noted that the function `util:eval()` is not native XQuery or XPath. The function could be rewritten without using `util:eval()`. However,

```
declare function local:counts($hits){
  let $facets := ("$hits/location",
                  "$hits/quantity",
                  "$hits/@featured")
  let $labels := ("locations",
                  "quantities",
                  "featured")
  return
    <facets>{
    for $facet at $p in $facets
    let $facetvalues := util:eval($facet)
    return
      <facet name="{$labels[$p]}">{
      for $a in distinct-values($facetvalues)
      return
        <value name="{$a}">
          {count($facetvalues[. eq $a])}
        </value>
    }</facets>
};
```

Listing 1: Naive Method

```
declare function local:cb($term, $data,){
  <value name="{$term}">{$data[1]}</value>
};

declare function local:counts($hits){
  let $cb := util:function(xs:QName("local:cb"), 2)
  let $facets := ("$hits/location",
                  "$hits/quantity",
                  "$hits/@featured")
  let $labels := ("locations",
                  "quantities",
                  "featured")
  return
    <facets>{
    for $facet at $p in $facets
    let $vals := util:eval($facet)
    return
      <facet name="{$labels[$p]}">
        {util:index-keys($vals, "", $cb, 10000)}
      </facet>
    }</facets>
};
```

Listing 2: Rvdb Method

we prefered sticking to the original as it was proposed by Van den Branden (2010).

4.2 Rvdb Method

The *Rvdb Method* was recently introduced by Van den Branden (2010). It is a method that exploits eXist-db specific range indexes. We list the source in Listing 2.

The main feature of this method is the use of the `util:index-keys()` function, this function causes all index-keys defined on a given node sequence to be reported to a callback function.[3]

[3]See `http://demo.exist-db.org/exist/functions/util/index-keys`

4.3 Indexed Method

Our, supposedly fast, indexed method is different from the other two methods in that it is not implemented in XQuery, but in Java instead. Thus, we can not list the XQuery source code, we describe our implementation instead. Our implementation makes heavy use of Bobo-Browse (Wang, 2010), which is "a Faceted Search implementation written purely in Java, an extension of Apache Lucene". This fits in naturally with eXist-db since that database is already using Apache Lucene under the hood for its full-text index. So, essentially, what we are doing is extending the full-text search engine of eXist-db with Faceted Search capabilities.

4.3.1 Indexing

Our indexer starts by looking at the `collection.xconf` file, as described in Section 2.3 (p. 5) and adds a Bobo-Browse FacetHandler for each facet it encounters there. The eXist-db database uses a SAX (Simple API for Xml, Megginson, 1997) parser when indexing. All xml files that are added to the database are streamed through all indexers node by node. An indexer can listen to this stream of nodes and act as it wishes. In our case, we simply add a Lucene 'document' with fields for each facet value every time we encounter a node we are interested in.

4.3.2 Querying

When querying our index, we will need an efficient way to calculate counts. Others have gone ahead (O'Neil, 1989) and introduced the use of bit-vectors. One bit-vector for each facet-value pair, with each bit in those vectors represents a document[4]. The bit for a document is turned to 1 for a facet-value if the facet-value is present in the document. These bit-vectors can be prepared at index time. Then, when a full-text query is fired a bit-vector can be constructed for the resulting document set. Calculating the counts for each facet-value then comes down to intersecting this last bit-vector with all other bit-vectors and calculating the cardinality. This functionality has been built in to Bobo-Browse.

4.3.3 Calculations with Bit-Vectors

Behind the scenes, Bobo-Browse uses efficient algorithms to deal with the facets. Nievergelt and Hinrichs (1993, ch. 8.3) describe a way to calculate the cardinality —the facet-value count— of a bit-vector efficiently. Instead of it being an $O(n)$ operation, with n the length of the bit-vector, it can be performed in $O(\log_2 n)$ by using a divide and conquer algorithm that exploits the capability of a processor to process a *word*[5] in parallel. The algorithm does so by viewing a *word* as a bit string, and recursively breaking this string into two parts until all bit strings are of length 1. Then, recursively summing up the parts leads to the bit sum; the cardinality. The real benefit comes from the fact that this summing up can be done in parallel when the *words* are lined up properly using bit-shifts.[6]

[4] Where a 'document' in our XML setting could be a smaller unit; an arbitrary XML node.

[5] In our experiments we use *words* consisting of 64 bits, the size of the registers of the processor we use, as it is implemented by the java.util.BitSet.

[6] See Nievergelt and Hinrichs (1993, p. 76) for details.

This same parallel capability is also exploited for the *bitwise-and* operations; both bit-vectors are split up in *words* and corresponding *words* from both vectors are *and*-ed together in parallel, leaving us with $\frac{n}{s}$, with s the size of the *word*[7], operations instead of the n operations of a naive implementation.

When the number of facet-values goes up, the significance of these optimizations becomes more and more apparent.

4.3.4 Efficient Mappings

While we get much of the implementation for free by using Lucene in combination with Bobo-Browse, it also leaves us with a difficulty. Lucene is designed in such a way that it assigns its own identifiers to documents it receives for indexing. The original eXist-db implementation of full-text search —that also uses Lucene— solves this by storing the eXist-db xml node identifiers into the Lucene index as fields. We do the same, so querying or drilling down is not a problem; the translation from a Lucene result into a sequence of xml nodes is trivial; it is a matter of requesting the stored field. The other way around, however, from a sequence of xml nodes to Lucene document-identifiers is not possible this way. Wang (2009) encountered the same problem while working on the LinkedIn data and implemented a solution into Zoie (Wang, 2008). We plan on using Zoie for this purpose but presently we use a very basic approach. We query the whole Lucene index and afterwards filter out all irrelevant nodes in the context of our query.

5 Experiments

5.1 Experimental Setup

We run our experiments on a machine with an `Intel Pentium Dual-Core 2.00GHz` processor with 2GB of memory running Fedora Core 8. The eXist-db version we use is a snapshot of the trunk made on November 5th 2010,[8] run with Sun Java 1.6.0_16 without adjusting any (memory) settings except for the `-Xmx1500m` flag to avoid running out of heap space.

Furthermore, we use XCheck (Afanasiev et al., 2006) as a wrapper around our experiments. Each experiment is run 4 times and the average runtime is taken over the last 3 runs, to avoid measuring any possible warm-up times. All times are measured in seconds, and we report total times, so including time spent on serialization as this is handled by eXist-db and should therefore be equal for all methods.

5.2 Scaling

To find how each of the three methods scales, we have run Scenario A and B for all datasets listed in Table 1 (p. 6). We present the average processing time over all steps (from both scenarios) per method per datasize in Figure 3 (following page). Detailed results per step, for each of the three methods are in Figure 4 (p. 12) and 5 (p. 13) for Scenario A and B respectively.

[7] Again, 64 bits in our experiments.

[8] Our exact code can be found at: `http://exist.svn.sourceforge.net/viewvc/exist/branches/anneschuth/`

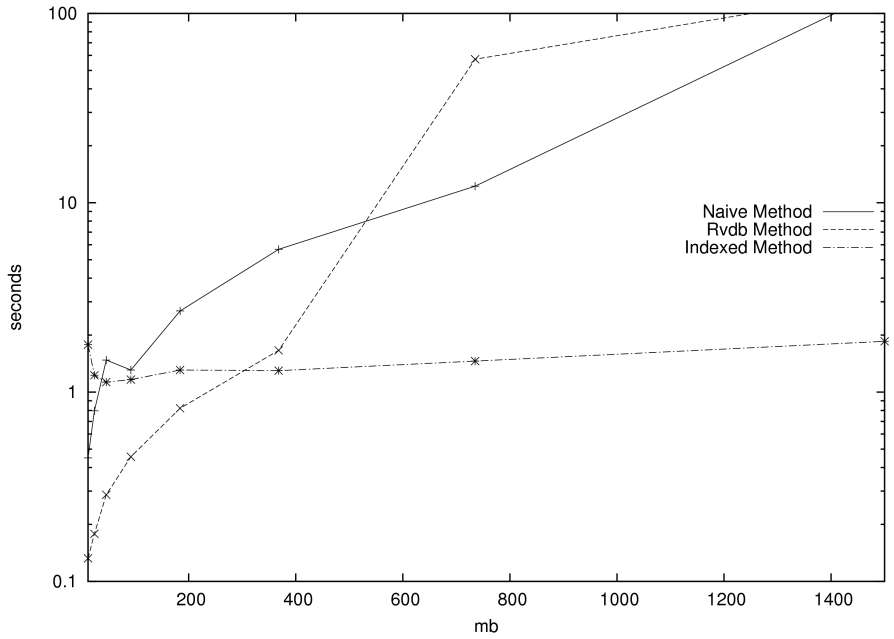

Figure 3: The average processing time over all steps (from both scenarios) per method for a growing dataset. Note the log scale of the vertical axis.

Looking at Figure 3, we see that initially the *Indexed Method* performs worse than both other methods but remains near constant while the time taken by both the *Naive Method* and *Rvdb Method* is polynomial. For datasets larger than about 400MB it is preferable to use the *Indexed Method*. It should also be noted that an average processing time of about 1 second per query is generally not acceptable in a user interface.

Both Figure 4 (following page) and 5 (p. 13) illustrate how the first drill-down step —step 2 in each scenario— for the *Indexed Method* consequently takes a lot of time. While all other timings for that method stay below or around 0.2 seconds, step 2 goes up to 8 seconds. We are not able to pinpoint the cause of this behavior yet. We *do*, however, point out that when we solve this issue the method will become within a very acceptable range with respect to processing time.

If we leave our *Indexed Method* aside, it is interesting to see that the *Rvdb Method* outperforms the *Naive Method* method for the earlier steps and vice versa for the later steps.

6 Conclusions

We conclude that the bit-vector implementation is the only one which scales with the datasize; the processing time stays near constant. Such behavior is a very desirable one in many applications. Our average running time of around 1 second for all queries on all datasets (going up to almost 300K nodes or 1.5GB) is promising but not yet good enough.

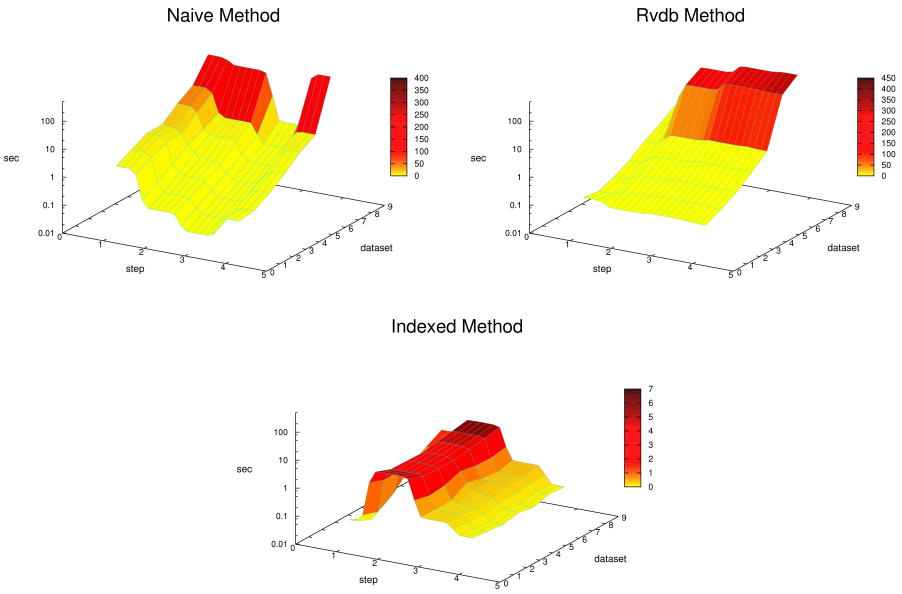

Figure 4: Processing time in seconds per step of Scenario A, for different datasizes, see Table 1 (p. 6). Figures are for the *Naive Method*, *Rvdb Method* and the *Indexed Method*. Note the log scale of the vertical axis.

The development of the faceted search micro-benchmark based on XMark and its application using XCheck proved useful. The plots give quick insights in the scalability of the algorithms, possibilities for improvements and anomalies in the code. A future version of the Benchmark could include a way to measure scalability over the number of facet-value pairs.

6.1 Future Work

There are two obvious starting points for improvements: find the reason for the outlier in our experiment, as mentioned in Section 5.2 (p. 10). Secondly, as mentioned in Section 4.3.4 (p. 10), we should implement a solution like Wang (2009) for fast mappings to and from positions in a bit-vector to node-identifiers probably by integrating Zoie.

Besides improvements in our implementation of the *Indexed Method*, we would like to extend our experiments. It would be insightful to investigate the behavior of the three methods when we vary the number of facet-values.

Also, a more precise measurement of where exactly in which method time is spent would give a clear indication of where we could still gain something.

And lastly, we did not mention the behavior of the methods under (heavy) updates of the dataset; we assumed a static dataset where documents are never removed or added. It is very likely that for such a scenario another method should be prefered.

Acknowledgements Maarten Marx acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh

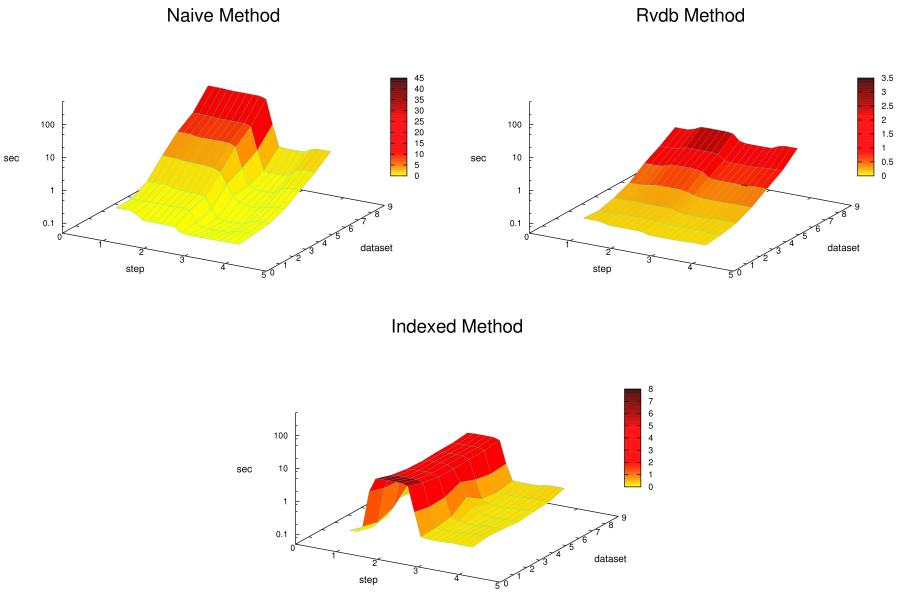

Figure 5: Processing time in seconds per step of Scenario B, for different datasizes, see Table 1 (p. 6). Figures are for the *Naive Method*, *Rvdb Method* and the *Indexed Method*. Note the log scale of the vertical axis.

Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599. This research was supported by the Netherlands organization for Scientific Research (NWO) under project number 380-52-005 (PoliticalMashup).

References

Afanasiev, L., Franceschet, M., and Marx, M. (2006). XCheck: a platform for benchmarking XQuery engines. In *Proceedings of the 32nd international conference on Very large data bases*, page 1247–1250.

Afanasiev, L., Manolescu, I., and Michiels, P. (2005). MemBeR: a micro-benchmark repository for XQuery. *Database and XML Technologies*, page 144–161.

Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., and Siméon, J. (2007). XML path language (XPath) 2.0, W3C recommendation 23 january 2007.

Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J., and Simeon, J. (2007). XQuery 1.0 an XML query language, W3C recommendation 23 january 2007.

Burke, R. D., Hammond, K. J., and Young, B. C. (1996). Knowledge-based navigation of complex information spaces. In *Proceedings of The National Conference On Artificial Intelligence*, volume 462, page 468.

Clark, J., Murata, M., et al. (2001). Relax NG specification. *OASIS Committee Specification*, 3.

Dash, D., Rao, J., Megiddo, N., Ailamaki, A., and Lohman, G. (2008). Dynamic faceted search for discovery-driven analysis. In *Proceeding of the 17th ACM conference on Information and knowledge mining - CIKM '08*, page 3, Napa Valley, California, USA.

English, J., Hearst, M., Sinha, R., Swearingen, K., and Yee, K. P. (2002). Hierarchical faceted metadata in site search interfaces. In *CHI'02 extended abstracts on Human factors in computing systems*, page 628–639.

Hearst, M. (2006). Design recommendations for hierarchical faceted search interfaces. In *ACM SIGIR Workshop on Faceted Search*, page 1–5.

Hearst, M. (2008). Uis for faceted navigation: Recent advances and remaining open problems. In *Proc. 2008 Workshop on Human-Computer Interaction and Information Retrieval*.

Hearst, M. (2009). *Search user interfaces*. Cambridge Univ Press.

Hearst, M., Elliott, A., English, J., Sinha, R., Swearingen, K., and Yee, K. P. (2002). Finding the flow in web site search. *Communications of the ACM*, 45(9):42–49.

Megginson, D. (1997). Sax: The simple api for xml. `http://www.megginson.com/SAX/`.

Nievergelt, J. and Hinrichs, K. H. (1993). *Algorithms and data structures: with applications to graphics and geometry*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

O'Neil, P. (1989). Model 204 architecture and performance. *High Performance Transaction Systems*, page 39–59.

Sacco, G. M. and Tzitzikas, Y. (2009). *Dynamic taxonomies and faceted search: theory, practice, and experience*. Springer-Verlag New York Inc.

Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I., and Busse, R. (2002). XMark: a benchmark for XML data management. In *Proceedings of the 28th international conference on Very Large Data Bases*, page 974–985.

Van den Branden, R. (2010). As a matter of fac(e)t: (mimicking) faceted searching in eXist. `http://rvdb.wordpress.com/2010/10/06/mimicking-faceted-searching-in-exist/`.

Wang, J. (2008). Zoie. `http://sna-projects.com/zoie/`.

Wang, J. (2009). Inverted index: Lucene docid, UID mapping and payload. http://invertedindex.blogspot.com/2009/04/lucene-dociduid-mapping-and-payload.html.

Wang, J. (2010). Bobo-Browse. `http://code.google.com/p/bobo-browse/`.