Text and Formula Search on ArXiv Documents

by

Corneliu C. Prodescu

Master Thesis in Computer Science

Prof. Michael Kohlhase
Name and title of first reviewer

Prof. Jürgen Schönwälder
Name and title of second reviewer

Date of Submission: August 23, 2014

Jacobs University
School of Engineering and Science

Declaration
I, Corneliu-Claudiu Prodescu, hereby declare that I have written this thesis independently, unless where clearly stated otherwise. I have used only the sources, the data and the support that I have clearly mentioned. This thesis has not been submitted for conferral of degree elsewhere.

Corneliu-Claudiu Prodescu
Bremen, August 23, 2014

Acknowledgements

First and foremost, I would like to express my uttermost gratitude to my supervisor Michael Kohlhase for continuous guidance and encouragement. Next, I would like to acknowledge work in the MathWebSearch system and code reviews from Radu Hambasan and work on the MathWebSearch user interface by Tom Wiesing. I am thankful for technical support provided by LaTeXML maintainer Deyan Ginev and constant server admin support from our system administrator Mihnea Iancu.

Finally, having a choice in life is a luxury. To my parents for granting me such a luxurious life, I am grateful.

Abstract

Formula search is a research topic tackled by a number of systems, with various approaches (looking for similar or exact matches), and on different scales (from few thousands to hundreds of millions of formulae).

MathWebSearch is an exact formula search engine based on unification. It provides a complete toolchain for crawling, indexing and querying mathematical expressions in Content MathML format. TeMaSearch is a system which combines MathWebSearch and a text search system (ElasticSearch) to provide simultaneous formula and text queries.

In this thesis we design and implement an improved TeMaSearch system, suitable for large corpora. Furthermore, we bring improvements across all MathWebSearch tiers, from core index compression, to data analytics, purification and canonicalization, to compression of the core index and, finally, to the user interface.

The goal of this is to make MathWebSearch and TeMaSearch feasible for deployment on large scale corpora, like the Cornell e-Print Archive (ArXiv), which contains close to 1 million documents.

Keywords

Formula Search, Content MathML, Search Systems Integration, MathWebSearch, ElasticSearch, Efficient In-memory Data Structures.

Contents

1 Introduction

As the world of digital information grows, being able to quickly retrieve information of interest is one of the most important tasks. Hence, search interfaces have become gateways to digital libraries.

While word search is successful for the world wide web (Google [15], Bing [5]), it is not enough for digital libraries with scientific content. As STEM[1] documents store key information in mathematical formulae, which are poorly indexed by text search engines, formula search is required. One STEM library which would benefit from formula search is the Cornell e-Print Archive (ArXiv) [4]. This corpus contains 922,701 LaTeXdocuments from various STEM domains: Physics, Mathematics, Computer Science, Quantitative Biology, Quantitative Finance, and Statistics [4].

Searching mathematical formulae comes with challenges different from its text counter part. Text adheres to a language and a dictionary which (up to some ambiguity) provides meanings. In contrast, formulae have an abundance of variables and notations which have a context sensitive meaning. For example, imagine the multitude of meanings p can take: p as a prime, $p(x)$ as a polynomial, $p(x)$ as a probability.

While formula search is tackled by a number of research projects [18, 20, 25], none can do exact formula search on large scale corpora (millions of LaTeX documents with billions of formulae) like ArXiv. Our goal is to develop such a system.

This thesis is structured as follows. In section 2 we present the state of the art in formula search and what are the tradeoffs of each system, in terms of scale and capabilities. Next, in section 3 we describe in-depth the systems which constitute a base for our work. Section 4 presents the system we have designed and implemented. In section 5, we evaluate the characteristics of our implementation. We conclude in section 6 and discuss future work in section 7.

2 State of the Art

Most formula search engines index formulae found in the MathML format. MathML is an open standard for representing Mathematics on the

[1]Acronym referring to the fields of Science, Technology, Engineering, and Mathematics

web. It is based on XML and comes in two flavors: presentation (human renderable) and content (machine oriented). The Content MathML format provides semantic support, allowing an expression like $x + y$ to be represented as an application of the symbol $+$ onto two identifiers:

Listing 1: ContentMathML for $x + y$

```
<apply>
  <plus/>
  <ci>x</ci>
  <ci>y</ci>
</apply>
```

We distinguish 3 categories of formula search engines:

Exact search
The XML tree of the query formula is matched against the trees of indexed formulae

Similarity search through verbalization
Each formula is encoded as a group of words and then a text search system is used for query and indexing

Similarity search through feature selection
Each formula is analyzed and a vector of features is extracted. Results are defined by a distance threshold in the feature space.

MathWebSearch is an unification-based search engine for mathematical formulae. It indexes Content MathML and its query language is Content MathML augmented with free variables which can match any subformula. MathWebSearch builds an in-memory index of the Content MathML trees of indexed formulae and uses unification to match the query with all formulae in the index, thus retrieving only exact hits. In the NTCIR-10 information retrieval challenge [1], MathWebSearch indexed 100,000 documents of the ArXiv corpus, containing 297 million formulae (including sub-formulae) in an index of 100 Gb [18]. Overall, MathWebSearch provides fast and accurate formula search, with a powerful query language, at the price of large memory usage.

EgoMath2 is a similarity-based formula search engine based on feature extraction. It indexes the Presentation MathML in documents. Each formula is transformed to a canonical form, parsed in preorder, and a feature vector is extracted. The similarity between a query and an indexed formula is given by cosine similarity. To prune similarity candidates, clustering algorithms (K-means, self organized maps and agglomerative hierarchical clustering) are applied in the index. The EgoMath2 developers have done an experimental deployment which indexes the mathematical articles of Wikipedia, containing 258,404 unique formulae [25]. EgoMath2's similarity approach works well for simple formulae, but decays as the formulae get more complicated. Furthermore, it was never deployed on a scale similar to ArXiv, so this would not be a viable solution.

WebMIaS is a similarity-based formula and text search engine based on Apache Lucene [3]. Presentation MathML or Content MathML formulae are parsed using a custom tokenizer which generates verbalizations (Math tokens are transformed into words). There are then indexed into Lucene as words. Query proceeds in a similar fashion, with the Math formulae being verbalized before passing the query to Lucene. In the NTCIR-10 information retrieval challenge, WebMIaS indexed 100,000 ArXiv documents, finding 73.4 million formulae in an index of 30 Gb [20]. This system scales well, but suffers in accuracy and query times. For example, the query $1 + \frac{2}{n-3}$ returns hits like $1 + \frac{1}{n-1}$ and $2 + \frac{1}{n-1}$, while the query time are over 6 seconds. [34].

TeMaSearch is a simultaneous text and formula search engine. It combines the text search capabilities of ElasticSearch with the unification queries of MathWebSearch. Formula queries are first resolved by MathWebSearch into encoded identifiers which are then combined with the text query and passed to the text search engine. This leverages the powerful query language of MathWebSearch and the inherent ranking in the text search engine. A deployment of this system serves the Zentralblatt Math corpus containing 1.97 million documents and 60 million formulae (including sub-formulae). The instance uses 7 Gb of RAM for the MathWebSearch index and 17 Gb of disk space for the ElasticSearch index.

All in all, similarity-based formula search engines work well for simple formulae, but they lack a precise query language. Furthermore, their accuracy is significantly lower, as shown in the NTCIR-10 information retrieval challenge, where MathWebSearch obtained an accuracy of 18.7%, with all similarity-based engines obtaining under 6.25% [1]. On the other hand, MathWebSearch provides a precise, mathematical query language and implements unification, but an ArXiv deployment would be beyond the memory limits of any single server.

In the rest of this thesis, we describe the design and implementation of a system based on TeMaSearch which can be deployed on a corpus of the scale of ArXiv.

3 Preliminaries

This section introduces the software systems our implementation is based on. We will start by outlining our project goals, followed by presenting each of our system dependencies. These software projects may be used as-is (LaTeXML, Elastic Search) or modified (MathWebSearch, TeMaSearch proxy, MathWebSearch frontend) to fit our requirements.

3.1 Project Goals

As discussed in section 1, our goal is to develop a text and formula search at ArXiv (Cornell e-Print Archive) scale. The ArXiv corpus contains 922,701 TeX documents from various STEM domains: Physics, Mathematics, Computer Science, Quantitative Biology, Quantitative Finance, and Statistics [4].

Extrapolating the estimates from [18], it contains 1.4 billion formulae (including sub-formulae). As discussed in section 2, formula search has been addressed by a number of research projects, but never at this scale.

For our system, we have set the following end-user requirements:

1. the system should be able to respond to queries containing words, as well formulae
2. the user interface should provide, for each hit, document data with highlighted query words and formulae
3. results should be returned to the end-user in a timely manner (in the order of seconds)

3.2 LaTeXML

Most of the digital formulae are available in LaTeX format, since LaTeX is easy to write and widely popular among STEM researchers. Unfortunately, in this format, formulae are not suitable for search since they are hard to parse and lack mathematical structure. A common approach is to convert LaTeX formulae to MathML using LaTeXML.

LaTeXML [24] is an open source program which converts TeX to various formats of XML. It mimics TeX behavior, but producing XML instead of dvi. It provides a processing pipeline and a model which are both extensible. LaTeXML allows definition files, called bindings, which define the mapping between $\TeX$ constructs and the XML fragments to be created. $\LaTeX$ML features a post-processor which converts this XML into other formats such as HTML or XHTML, with options to convert the Math expressions into MathML (Content MathML and Presentation MathML).

3.3 MathWebSearch

MathWebSearch is an unification-based search engine for mathematical formulae [19, 22, 17]. MathWebSearch indexes formulae which are represented as Content MathML, which describes the structure of Mathematical expression as XML trees.

Each expression is encoded as a set of substitutions based on a depth-first traversal of its Content MathML tree. For example, $f(g(a,a),b)$, which is represented as shown in Listing 2, would be encoded as:

$$\begin{array}{lll}
@0 & \rightarrow @1(@2,@3) & \text{// <apply> of arity 3} \\
 & \rightarrow f(@2,@3) & \text{// <ci>f</ci>} \\
 & \rightarrow f(@4(@5,@6),@3) & \text{// <apply> of arity 3} \\
 & \rightarrow f(g(@5,@6),@3) & \text{// <ci>g</ci>} \\
 & \rightarrow f(g(a,@6),@3) & \text{// <ci>a</ci>} \\
 & \rightarrow f(g(a,a),@3) & \text{// <ci>a</ci>} \\
 & \rightarrow f(g(a,a),b) & \text{// <ci>b</ci>}
\end{array}$$

At each step, exactly one variable is substituted and this path of substitutions describes the initial formula. The MathWebSearch index is an in-memory trie of substitution paths and each leaf has an associated identifier. Each identifier is used to store occurrences and context about the respective formula in a database.

Listing 2: ContentMathML for $f(g(a,a),b)$

```
<m:apply>
  <m:ci>f</m:ci>
  <m:apply>
    <m:ci>g</m:ci>
    <m:ci>a</m:ci>
    <m:ci>a</m:ci>
  </m:apply>
  <m:ci>b</m:ci>
</m:apply>
```

To save space and improve lookups, Content MathML tokens are encoded as integer identifiers using a dictionary. The token dictionary starts empty and is populated with $token \rightarrow token\_id$ mappings while indexing documents.

MathWebSearch indexes (X)HTML documents. These are crawled by the MathWebSearch Crawler , which generates harvests. These contain extracted Content MathML formulae along with location information (XML identifier of the math element and an XPath to it - necessary for subexpressions). An example MathWebSearch harvest is provided in listing 3.

Listing 3: Example MathWebSearch Harvest

```
<mws:harvest>
  <mws:expr url="http://math.example.org/article1#f2.3">
    <m:apply>
      <m:ci>&#x2192;</m:ci>
      <m:ci>x</m:ci>
      <m:cn>0</m:cn>
    </m:apply>
  </mws:expr>
  <mws:expr url="http://math.example.org/article2#f4.2">
    <m:ci>a</m:ci>
  </mws:expr>
   ...
</mws:harvest>
```

MathWebSearch provides a RESTful HTTP API which supports queries in the form of XML. MathWebSearch queries use Content MathML augmented with variables *mws:qvar* which match any subterms. An example query is provided in Listing 4. This will retrieve the first 5 formulae which represent addition of two terms.

Listing 4: Example MathWebSearch Query

```
<mws:query limitmin="0" answsize="5">
  <mws:expr>
    <m:apply>
       <m:plus/>
       <mws:qvar>qvar_x</mws:qvar>
       <mws:qvar>qvar_y</mws:qvar>
    </m:apply>
  </mws:expr>
</mws:query>
```

3.4 ElasticSearch

ElasticSearch [10] is an open-source full text search engine built with scalability and availability in mind. It offers partitioning of data in shards, distribution across multiple nodes, replication and fault tolerance. Its core text search engine provides full-text queries with ranking, with the ability to define simple schemas and search using exact matches, wildcards, and range queries, among many others.

Indexing and querying in ElasticSearch are done using a RESTful HTTP API which accepts JSON. It supports typed data (text, integers, floats) and key-value pairs in JSON are used to support semi-structured indexing.

An example document is presented in Listing 5.

```
{
  "title" : "Example Document",
  "date" : "2014-01-01-093000Z00",
  "ids" : [2, 3, 5, ...],
  "body" : "This is an example document",
  ...
}
```

Listing 5: Elastic Search Document

ElasticSearch supports text queries within specific fields. Listing 6 presents a query for documents containing both words *example* and *document* in their *body* field.

```
"query" : {
  "match" : {
    "body" : {
      "query" : "document example",
      "operator" : "and"
    }
  }
}
```

Listing 6: Example ElasticSearch Text Query

Last but not least, ElasticSearch supports identifier queries within specific fields. Listing 7 presents a query for documents containing at least one of the identifiers $[1, 2, 3]$ within their *ids* field.

```
"query" : {
  "terms" : {
    "ids" : [1,2,3],
    "minimum_match" : 1
  }
}
```

Listing 7: Example ElasticSearch Identifier Query

3.5 TeMa Search

TeMaSearch is an experimental system built on top of MathWebSearch and ElasticSearch to provide simultaneous text and formula search. The system fulfills two use cases: indexing and querying.

TeMaSearch is designed to index a set of XHTML documents, which is achieved in two steps:

- XHTML documents are presented to MathWebSearch which indexes the formulae and generates JSON versions of them with annotated Math identifiers.
- JSON documents produced in the previous step are indexed in ElasticSearch.

The MathWebSearch indexing process is slightly modified to generate, for each indexed document, a JSON file, as presented in Listing 8. This represents the original document, augmented with identifiers and location information for each formula.

```
{
  "ids" : [
    6,
    7,
    ...
  ],
  "id_mappings" : [
    {
      "id" : 6,
      "xpath" : "\/*[1]\/*[2]\/*[2]\/*[4]\/*[1]",
      "url" : "5362108.xhtml#S0.Ex1.m1.1"
    },
    {
      "id" : 6,
      "xpath" : "\/*[1]\/*[2]\/*[2]\/*[1]",
      "url" : "5362108.xhtml#S0.Ex2.m1.1"
    },
    ...
  ],
  "xhtml" : ...
}
```

Listing 8: Elastic Search Harvest File

After indexing the original documents in MathWebSearch, the augmented JSON documents are indexed into ElasticSearch and the system is ready to receive queries.

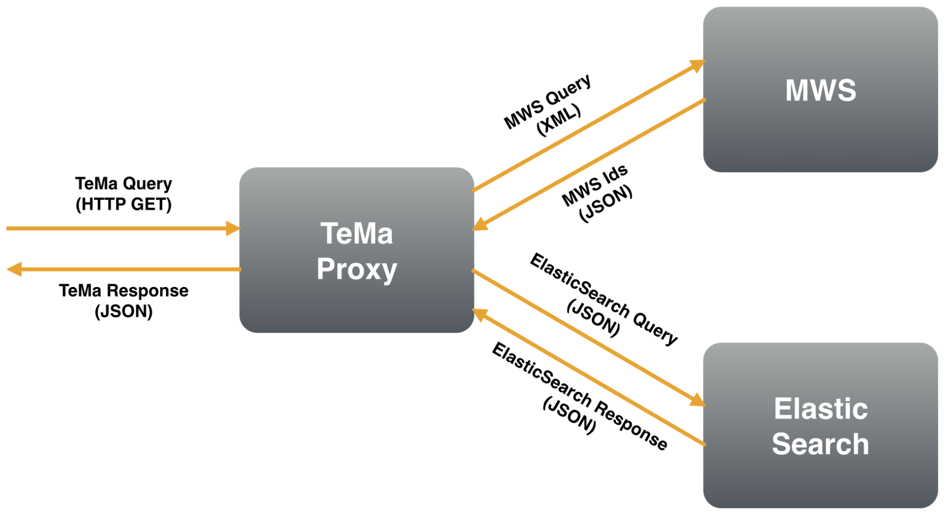


Figure 1: TeMa Search Query Architecture

The query architecture is presented in Figure 1. To manage requests, TeMaSearch has a proxy which exposes a RESTful endpoint. It accepts HTTP `text` and `math` query parameters and returns document hits as `JSON` data. Hits are defined at document level, since queries intersect multiple tokens (words and formulae). A document is considered a hit if it contains all words in the text query and at least one formula matching the math query.

Queries are run in a 2-step process. First, MathWebSearch is queried using the `math` query parameter and a set of identifiers representing matching formulae is returned. Next, the ElasticSearch query is composed using the `text` query parameter and the formula identifiers, as displayed in Listing 9. The `match` component searches for exact matches of all words in `text_query` in the `xhtml` of documents, while the `terms` component checks for at least 1 occurrence of `formula_ids` in the `ids` of documents.

```
"query" : {
    "bool" : {
        "must" : [
            "match" : {
                "xhtml" : {
                    "query" : <<text_query>>,
                    "operator" : "and"
                }
            },
            "terms" : {
                "ids" : <<formula_ids>>,
                "minimum_match" : 1
            }]
    }
}
```

Listing 9: Elastic Search Harvest File

Finally, the results from ElasticSearch, including the matching documents, are sent as response to the TeMa Proxy query.

3.6 Other Tools

All software components we implemented were developed under revision control, using git [11] and hosted as GitHub [12] projects. We used Waffle.io [33] on top of GitHub issues to track work progress. Development was done in the agile paradigm, with sprints of 2 weeks. Tasks were assigned per sprint and unit tests were written based on remaining time within the respective sprint.

A coding style derived from the Google C++ Code Style [14] was imposed using the tool clang-format [8]. Major feature commits were code reviewed using the Phabricator Differential [30] system.

Throughout development, we used a number of language specific tools. For C/C++, we used the CMake [9] build system, a number of sanitizer tools (Valgrind [32], Clang Address, Memory, Thread Sanitizers [7]) and the debugger (gdb [13] on Linux, lldb [21] on Mac). For Javascript, we used the NodeJS system [27] along with the Node package management tool npm [26].

4 Implementation

4.1 Design Overview

Since TeMaSearch (3.5) already fulfills our end-user project goals described in section 3.1, we started modeling our design after it.

After running a small scale experiment with 500 documents, using the original TeMaSearch system, we detected a number of issues:

1. the generated Math index was 0.3Gb, which means we would need 600Gb or RAM for 1 million documents, assuming linear memory increase.
2. indexing took 106 seconds, which entails 59 hours for indexing 1 million documents.
3. responding user requests with full documents is not an option since the system returns 2Mb of data per document, and rendering these takes several minutes.

Analyzing these issues, it became clear that our implementation should optimize the MathWebSearch index (1). Furthermore, indexing should become persistent, as re-indexing at every restart is not acceptable (2). Finally, (3) clearly shows that we need to change the TeMaSearch architecture to generate document snippets.

We will tackle MathWebSearch index memory optimizations in section 4.2 and persistency in section 4.2.3. Next, we will discuss hit units and document snippets (4.3). Designing with document snippets in mind requires changes to the document crawler (4.4), as well as the ElasticSearch schema (4.5). Next, we will leverage the new MathWebSearch index to build formula analytics (4.6) and deal with change management (4.7). Using lessons learned from analytics, we will discuss formula purification techniques (4.8). Last but not least, we will briefly describe our web user interface (4.9). Finally, we will present a short summary of our system (MathWebSearch, ElasticSearch, TeMaSearch proxy, web frontend) in section 4.10.

4.2 MathWebSearch Trie Index

As discussed in section 3.3, the central indexing data structure in MathWebSearch is a trie of depth-first substitutions. Here we will describe how we have improved its space and time efficiency.

4.2.1 Narrow Identifiers

Each internal node of the central index is a map from a pair of meaning identifier (64bit integer) and arity (32bit integer) to a new node.

$$(token\_identifier, arity) \rightarrow node$$

Each leaf has a formula identifier (64bit integer) and the hit counter (64bit integer) for the number of hits associated with the respective formula.

We started by analyzing how much of the identifier space was used. We run a few tests on a set of 500 ArXiv documents. The resulting index contained 4506 unique tokens and 218662 unique formulae, while the maximum number of hits per formula was 61786. The first two increase sub-linearly with index size, while the last one increases linearly with document count, hence we can safely assume that, for 1M ArXiv documents, these numbers shouldn't be above 9M, 437M, respectively 123M.

Following these assumptions, we reduced formula identifiers and hit counters to 32bit integers, token identifiers to 24bit integers and arities to 8bit integers. Having reduced the identifier space, we have leaf nodes of 8 bytes, while internal nodes use only 12 bytes (4 bytes for the token and arity and 8 bytes for the pointer to the next node) for every child mapping.

4.2.2 Space-Optimized Map

While leaf nodes seemed simple enough to be optimal (excluding data compression), we felt we could do better with the map data structure. The default STL [6] map implementation is a red black tree. As any tree data structure, it uses extra memory for pointers, one for each of items in the tree. Since our items are small (12 bytes), the overhead of a pointer (8 bytes) makes it 60% memory efficient.

STL offers an alternative map (unordered map), which is implemented as a hash table, but this incurs the same pointer memory overhead.

We decided to use an associative vector as a map implementation. This is a sorted vector of key-value pairs. Insertion and deletions are $O(n)$, while lookup operations are $O(\log n)$. The insertion high cost is paid only during indexing and currently we do not use deletions, so this remains optimal for our use case (we are lookup-biased). The associative vector has 2 advantages:

- no additional memory is lost on pointers
- sequential keys are accessed faster thanks to cache locality

While pointer memory overhead disappears, extra memory is used for performance reasons at the allocator level. A Vector is backed by a continuous region of memory holding enough space for n items. When one inserts the $n + 1$-th item, this region is grown using a multiplier x to capacity xn and the first n items are copied to the new region, before releasing the old one. Capacity is increased exponentially to mitigate the overhead of allocation and copy for large vectors. However, as a result of this, we have, on average, $(nx - n)/2$ unused item slots in a container of size nx. This translates to an efficiency of $\frac{n+(nx-n)/2}{nx} = \frac{x+1}{2x}$. Note that, STL uses $x = 2$, entailing an efficiency of 75%.

4.2.3 Persistent Index

Since the indexing operation is time consuming, we decided to develop a persistent index data structure.

As design constraints, we imposed the following:

persistency
: The index should survive, without corruption, process restarts and crashes, as well as machine reboots and power losses.

space efficiency
: The persistent index should, when loaded into RAM, be at least as efficient as the temporary index.

loading efficiency
: The index should load significantly faster than indexing from harvests.

We decided to write the entire trie data structure, node by node, into a file and load it by memory mapping the file. While this works for absolute data (arities, token identifiers, etc), it doesn't work for relative data (pointers). To mitigate this, we transformed pointers into offsets from the beginning of the file.

We implemented a callback iterator which exposes the index trie in depth-first order, with callbacks for pushing and popping nodes. This is used to process the trie in post-order[2]. Trie nodes are written in post-order, since writing a node requires the file offsets of its children to be known. During the process, $O(dn)$ extra-memory is used to store children offsets, where d is the depth of the trie and n is the average branch factor (number of children a node has). Note that this is significantly lower than the size of the index $O(n^d)$.

Furthermore, for the saved internal nodes, we use children maps of optimal capacity, thus mitigating the fill factor issue described in section 4.2.2. Note that this only compresses the saved index, not the one built in RAM during indexing.

This optimizes the persistent index, but its size remains capped by the maximum size temporary index that we can build. We will address this, by building smaller persistent indexes and merging them, as described in section 4.7.

4.2.4 Pointer Compression in Persistent Index

Next, we looked at how we can narrow the pointers (offsets) inside the saved index. 64bit pointers are generally wasteful, since systems do not come close to $2^6 4$ bytes of addressable space. In fact, according to the current 64bit specification (AMD64 [2]), address translation hardware will only use the 48 least significant bits of an address.

We decided to narrow offsets to 32 bits. This reduces children map items from 12 bytes to 8 bytes, but limits the amount of addressable memory to 4Gb. To improve this limit, we introduced 8byte alignment for all nodes saved to the index file and decided to address only such offsets. Once the file is mapped into memory, the real address is computed as $file\_start\_address + 8 \times offset$. This improves the maximum size of a saved index to 32Gb.

Since we had available a machine with 96 Gb, we decided we needed to do even better. Firstly, we decided to change the offset reference point: instead of using the start of the file, we used the current node. Then, we decided to implement 2 types of internal nodes: one with 8byte offsets

[2]In post-order traversal, child subtrees are processed before the current node

(used for nodes which have children farther than 32Gb away - an obvious example is the root for tries bigger than 32 Gb) and one with 4byte offsets (for all other internal nodes). Finally, the formula to compute the real address of a child node is $node_address - 8 \times offset$.

4.2.5 Summary

We have created a serializable trie index which can be mapped into memory. It has a fixed size header (magic number, version, checksum, file offset of the root node), followed by serializations of the trie nodes. The nodes are one of:

long internal node
This node stores mappings from pairs of $(token\_identifier, arity)$ and 64bit offsets to the child node. The offsets are relative to the current node and negative (child nodes are written before parent nodes). This types of nodes have a memory footprint of $8 + 12c(+4)$ bytes, where c is the number of children The 4 bytes are possible alignment padding to maintain the 8 byte alignment.

internal node
This node is similar to the long internal node, with the exception that it uses 32bit offsets. Its memory footprint is $8+8c$ bytes, where c is the number of children

leaf node
This node stores a formula identifier (32bit integer) and the hit counter (32bit integer) for a memory footprint of 8 bytes

In section 5.1 we will analyze the characteristics of this implementation step by step.

4.3 Hit Unit and Snippet Generation Analysis

MathWebSearch considers hits at formula level. For example, if a document contains $x + y$ twice, the query $x + y$ will return both of these, as different hits. Of course, this doesn't make sense when the query consists of formula and text, since no single formula can be a hit for text. TeMaSearch gathers formulae within a document in a single hit unit. As such, a query of $x + y$ would return the full document as a single hit and the two instances of $x + y$ would be highlighted. This works well for small documents, like the ones in the Zentralblatt Math corpus, as users can immediately grasp the content of the document and the formulae.

As described in 4.1, since ArXiv documents are typically large, showing full documents is not practical. In our initial experiment, retrieving full documents generated 4Mb of data per request[3].

The large volume of data per document makes it clear that snippets are necessary. Furthermore, these need to be generated on the backend, thus avoiding large data transfers (to the proxy or the user interface).

ElasticSearch has a snippet generation system and provides a query API for it. However, since it is designed for plain text, it generates invalid HTML. Following this, we decided to switch our ElasticSearch schema to plain text documents.

To mitigate this issue, we have improved the MathWebSearch document crawler to extract plain-text, along with linked MathML data (4.4). Following this, we have changed the TeMaSearch document schema to pass this information along (4.5).

4.4 Document Crawler

As described in section 3.3, the MathWebSearch crawler reads (X)HTML documents and generates MathWebSearch harvests.

We have changed this document crawler on two aspects. Firstly, we have augmented the harvest format to allow arbitrary data to be associated with each document (or expression). Secondly, we have updated the crawler core to automatically strip non-Math HTML/XML content (transforming it to plain-text) and introduced a configuration language which allows metadata crawling.

4.4.1 MathWebSearch Harvest v2

The new MathWebSearch harvest format consists of 2 types of elements: expressions (expr) and data, both defined in the MathWebSearch namespace. An example harvest is provided in Listing 10.

```
<harvest>
<data data_id="0">
```

[3]These was a request for a page of results containing 5 documents along with some metadata

```
  ... document specific data ...
</data>
<expr url="math1" data_id="0">
  <apply>
    <geq/>
    <ci>m</ci>
    <cn>2</cn>
  </apply>
</expr>
...
</mws:harvest>
```

Listing 10: Example MathWebSearch Harvest v2

The new version adds data elements which contain arbitrary XML data and have the data_id attribute, an unique identifier which represents the respective data. Expression (expr) elements have the same specification as in the previous harvest version, with the addition they can have the data_id attribute which link expressions to their respective data elements. We have chosen this format since it allows linking multiple expressions to the same data. As a plus, this maintains backwards compatibility to the old format, where data elements and data_id attributes do not exist.

4.4.2 Crawler Engine

The crawler engine was extended to support (X)HTML documents generated by various systems. For this purpose, a small configuration language was created. It specifies what data should be extracted from the document (along with Math formulae), XPaths [35] for the document identifier, as well as user-defined metadata elements (title, author, etc.).

Furthermore, a text-with-math XPath can be specified to generate stripped (X)HTML. The resulting plain text has references (in the form *math«id»*) in place of Math formulae, which are saved separately. This is useful for two reasons:

- the text indexing service we use down the pipeline (ElasticSearch) is (X)HTML agnostic. As a consequence, it would wrongly index tags, and it would offer snippets which would be invalid (X)HTML.
- space is saved, as the original tags (which would need to be XML-escaped) are not stored in the harvest

An example configuration file is provided in Listing 11.

```
{
    "shouldSaveData" : true,
    "textWithMathXpath" : "//body",
    "documentIdXpath" : "",
    "metadata" : {
        "title" : "//title",
        "author" : "//span[@class='ltx_authors']"
    }
}
```

Listing 11: MathWebSearch Crawler Configuration

The harvested document data is saved inside the <data> elements of the MathWebSearch harvest using the following schema:

data/text
: plain text generated from stripping (X)HTML tags

data/id
: Unique document identifier

data/metadata/<field>?
: User defined metadata fields

data/math*
: original MathML of formulae found in (X)HTML content which was stripped

An example harvest file highlighting this schema is provided in Listing 12.

```
<harvest>
<data data_id="0">
  <id>ntcir-11-math2-wiki/00065.html</id>
  <metadata>
    <title>Quantum Interpretation of ...</title>
    <author>John Doe</author>
    <zbl_id>0343.2322</zbl_id>
  </metadata>
  <text>Under the constraint math1, the formula ...</text>
  <math local_id="1"> ... </math>
  ...
</data>
<expr url="math1" data_id="0">
```

```
  ...
</expr>
...
</harvest>
```

Listing 12: Example Harvest data

4.5 TeMaSearch Schema

Elastic Search indexes documents in the form of JSON, allowing a flexible schema. Since, its tokenizers are tailored for plain text documents, we did a bit of preprocessing. We replaced all MathML nodes with identifiers prefixed with the word math (for example, *math42*) and then we stripped all HTML and XML tags.

The JSON schema used to index a document has 4 components:

- metadata
- formula identifiers corresponding to the formulae in MathWebSearch
- preprocessed plain text
- math elements mappings from prefixed math identifiers to the original MathML formulae.

An example document is presented in Listing 13.

```
{
  "metadata" : {
    "title" : "An alternative formula for ...",
    "author" : "John Doe"
  },
  "math_ids" : [1, 3, 6, ...],
  "text" : "Under the constraint math1, the formula ...",
  "math_elements" : {
    "math1" : "<math id=\"math.19064.0\" alttext=\" ...",
    ...
  }
}
```

Listing 13: Example Elastic Search Document

4.6 Index Analytics

The efficiency of the load-able index can be leveraged to do analytics on the mathematical expressions in a corpus.

Imagine having to determine the number of *trivial* expressions in a corpus. We define as trivial all expressions which have depth less than 2 in Content MathML representation. For example, x and $x + y$ are trivial, while $\frac{x+y}{z}$ is not. This would normally require processing all documents, extracting the Math expressions, analyzing which expressions fit the triviality rule and then aggregating the counters.

Having the MathWebSearch index saved, this can be done trivially, by loading the index and running the algorithm on one data structure, in the RAM of one machine.

We have built an analytics interface which allows programatic access to the index. A user needs to provide C++ implementation for 3 methods:

analyze_begin
This is called before any expression is analyzed

analyze_expression(expression, num_hits)
This method is called for every expression in the index, with the number of times it was found in indexed documents (num_hits).

analyze_end
This is called after all expressions have been analyzed

```
static int total_hits;
static int unique_hits;
static double depth_sum;
static double size_sum;

AnalyticsStatus analyze_begin() {
    return ANALYTICS_OK;
}

AnalyticsStatus analyze_expression(CmmlToken* cmmlToken,
                                   uint32_t num_hits) {
    if (verbose) {
        printf("Analyzing␣expression␣%s\n",
               cmmlToken->toString().c_str());
    }
    total_hits += num_hits;
    unique_hits++;
    depth_sum += cmmlToken->getExprDepth();
    size_sum += cmmlToken->getExprSize();
    return ANALYTICS_OK;
}

void analyze_end() {
    printf("%d␣hits\n", total_hits);
    printf("%d␣unique␣expressions\n", unique_hits);
    printf("Average␣depth␣is␣%f\n", depth_sum / unique_hits);
    printf("Average␣size␣is␣%f\n", size_sum / unique_hits);
}
```

Listing 14: Example Analytics Program

An example analytics program is presented in listing 14. It computes the number of expressions in the index (unique and total), as well as average expression depth and size.

Using the analytics infrastructure is straight-forward. The build system picks up source files in the analytics directory. These are compiled into analytics executables which can run on saved indexes. In our experiments, running the job in listing 14 on an index of 100k documents (15Gb) took 1h20m.

4.7 MathWebSearch Index Merging

We have implemented a tool which merges MathWebSearch indexes, for the following purposes:

Change Management
: It allows an user to index a new set of documents incrementally, on top of an existent index. This is achieved by indexing the new documents into a separate index and then merging it with the original index. This saves time, as the large set of initial documents does not need to be re-indexed.

Maximum Memory Index
: This improves the maximum size of a persistent index. As described in section 4.2, the persistent index is more optimized than the temporary memory index used to build it. Hence, with fixed RAM machine, one cannot serve a persistent index of maximum

size, as building it requires more RAM than it has available. This is mitigated by building 2 persistent indexes and then merging them.

As described in section 3.3, the MathWebSearch index state consists of 3 parts:

- token dictionary
- index trie
- formula database

We will described the merging algorithm for each component. We will call the first index the reference index and the second index the auxiliary index.

4.7.1 Token Dictionary

Token dictionaries are loaded into memory. We build another dictionary, called a translation dictionary. This is a mapping $token\_id \rightarrow token\_id$, which translates from auxiliary token identifiers to reference token identifiers. Note that in this process, we may also add elements to the reference token dictionary (for tokens which were not in the reference index). We save the reference token dictionary and keep only the translation dictionary.

4.7.2 Index Trie

Index Tries are loaded into memory with read-write permissions. The token identifiers in the auxiliary index are translated using the translations dictionary. As described in section 4.2, the maps in the index trie are actually sorted vectors of

$$((token\_id, arity), child\_relative\_offset)$$

This behaves as a $(token\_id, arity) \rightarrow child\_relative\_offset$ map by doing binary search on the sorted pairs in the vector. Hence, after $token\_id$-s are translated, in each vector the nodes are re-sorted. This is trivial, since each sort is done independently for each trie node and relative offsets do not need to change (as they are relative to the start of the node).

Now we have two index tries using the same token dictionary. Formula identifiers in leaves are still out of sync and we will address these in 4.7.3.

Similar to the approach from 4.2.3, we use depth-first iterators through the tries. Next, we have built a merged iterator on top of the two depth-first iterators (one through each trie). This advances, at each step through one of the tries, achieving a merged view of the them.

Finally, this merged iterator is used to process the 2 tries and write the nodes, in post-order, as a single persistent trie.

4.7.3 Formula Database

The databases are merged by copying data from the auxiliary database into the reference one. This is done while traversing the tries with the merged iterator. When it encounters a leaf, we distinguish 3 cases:

- The leaf belongs to the reference trie exclusively. Nothing needs to be done
- The leaf belongs to the auxiliary trie exclusively. A new formula id, different from any of the ones in the persistent index needs to be generated. The data associated with the old formula id needs to be copied to the reference database under the new formula id.
- The leaf belongs to both tries. Note that here, we will likely have two different formula ids, one from each index. The data in the auxiliary database under the auxiliary formula id needs to be appended to the data in the reference database under the reference formula id.

4.8 Expression Preprocessing

Real world corpora, composed of human-authored documents, express a diverse spectrum of Math expression and representations for mathematical expressions. While LaTeXML tries to canonicalize these representations, the current results are far from perfect.

A popular misuse of LaTeX is labeling text as *italic* by placing it into math mode. For example, $italic$ will be interpreted by LaTeXML as a multiplication of 6 identifiers, as shown in Listing 15.

Listing 15: Content MathML generated from $italic$

```
<m:apply>
    <m:times/>
    <m:ci>i</m:ci>
    <m:ci>t</m:ci>
    <m:ci>a</m:ci>
    <m:ci>l</m:ci>
    <m:ci>i</m:ci>
    <m:ci>c</m:ci>
</m:apply>
```

To get rid of these, we could come up with rules for rejecting some expression patterns[4], but this can remove false positives and is generally error-prone.

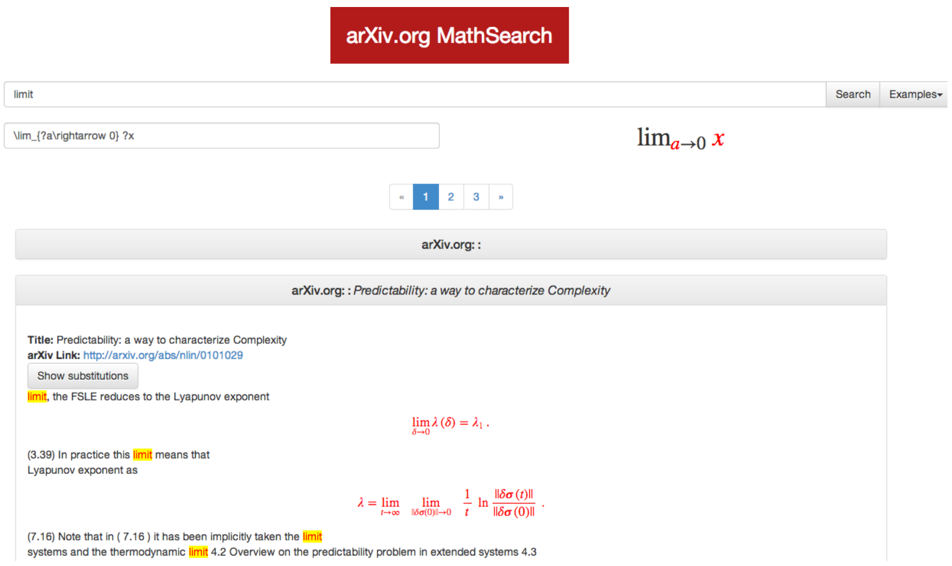
Instead, we decided to go for a more general solution. Since content identifiers (<ci>) are mathematical variables with no fixed value, their name does not carry any information (x, y, z can represent anything). As a result, we decided to canonically rename all <ci>s in any formula. This has several advantages:

- $a + b$, $b + a$, $x + y$ become equivalent.
- math-mode misuse is mitigated, many of the uses falling into one of a few classes (based on how many <ci>s they generate)

Note that, applying this to all <ci>s turned out to be an issue, as some notations are symbols, although they are inferred as <ci>s by LaTeXML (for example O in Landau Notation $O(n)$). We went for a tradeoff by only applying <ci> renaming to a predefined set of ascii characters.

We found another issue while running an analytics job which sampled scarce formulae[5]. As such, we noticed lots of <mtext> formulae, each one containing plain text sentences.

These are generated by text inserted in math equations using the \text macro. Obviously, these were not searchable as regular math in our index, so we decided to ignore mtext nodes while indexing. While occurrences of mtext were not very common (we found 480k such formulae in 100k documents), they had a huge impact on the token dictionary, which contained all 480k of them, among a total of 665k tokens.



$\lim_{a \to 0} x$

arXiv.org: :

arXiv.org: : *Predictability: a way to characterize Complexity*

Title: Predictability: a way to characterize Complexity
arXiv Link: http://arxiv.org/abs/nlin/0101029

limit, the FSLE reduces to the Lyapunov exponent

$$\lim_{\delta \to 0} \lambda(\delta) = \lambda_1 .$$

(3.39) In practice this limit means that
Lyapunov exponent as

$$\lambda = \lim_{t\to\infty} \lim_{\|\delta\sigma(0)\|\to 0} \frac{1}{t} \ln \frac{\|\delta\sigma(t)\|}{\|\delta\sigma(0)\|} .$$

(7.16) Note that in (7.16) it has been implicitly taken the limit
systems and the thermodynamic limit 4.2 Overview on the predictability problem in extended systems 4.3

Figure 2: MWS Search User Interface at http://jupiter.eecs.jacobs-university.de/arxiv-demo.

4.9 User Interface

For the user interface, we started with the TeMaSearch web interface and adapted it to present snippets instead of full documents.

As shown in Figure 2, it provides two input fields: one for text and one for LaTeX formulae. These fields become TeMaSearch proxy, providing a text and Math query and rendering the results. Formula preview is generated using the LaTeXML TeX $\rightarrow$ MathML conversion service.

The server sends snippets with highlight annotations (using a highlight HTML class). For formula highlights, additional XPath [35] data is provided, since the query hit can be a subexpression of a document formula.

4.10 Summary

Here we will summarize the components of our system and how they fall in place.

[4]For example, reject expressions which consist of a single multiplication of more than 5 content identifiers (<ci>)

[5]Formulae are scarce if they occur rarely in a corpus. In our case, we mean formulae which occur only once in the index

We use LaTeXML to convert LaTeX sources to HTML/XHTML. These documents are then processed by the MathWebSearch Crawler which generates harvests which represent the documents, in plain text, along with the formulae they contained. MathWebSearch reads the harvests and generates an index with all formulae.

Next, we use a MathWebSearch utility to annotate the formulae in the harvests with formula identifiers from the MathWebSearch index. These annotated documents are indexed into ElasticSearch under the schema presented in section 4.5.

Once this is done, the system is ready to receive queries. A modified version of the TeMaSearch Proxy (3.5) intermediates requests, querying MathWebSearch for math identifiers, and ElasticSearch for text and math identifiers. It exposes a RESTful HTTP API, on top of which we built an user web interface(4.9).

5 Evaluation

In this section, we will present an evaluation of our system. We will start by analyzing the memory optimizations we have implemented on the MathWebSearch index (5.1). Next, we will examine the advantages of the persistent index (5.2). Finally, we will look at how our system performed in the NTCIR information retrieval challenge (5.3).

5.1 MathWebSearch Index Memory Optimizations

5.1.1 Setup and Instrumentation

For this part of the evaluation we have randomly selected 500 documents, with a total size of 359Mb. These were selected from the ArXMLiv no-errors set (ArXiv documents which are converted without errors by LaTeXML). After running the MathWebSearch crawler, we generated 533 Mb of harvests containing over 1.3 million formulae (including subformulae). We will refer to this as *eval-dataset-1*.

To evaluate each index optimization incrementally, we prepared commits which disable each one individually.

We measured 3 metrics:

Total Memory Usage

This is the total memory footprint of the process. This is measured by reading the amount of resident memory the process uses using utilities like top.

Total Index Size

This is the size of the index data structure in memory. Since this is a C++ data structure with multiple indirection levels (pointers), computing this is hard. After a little research, we turned to the memory management analyzer valgrind, which already instruments all allocations and tracks address indirections. To obtain the desired result, we instrumented our code to leak the index data structure, as this will get reported, along with the leaked size by valgrind. Note that there are more comprehensive heap profiler tools, like gperf, but in our use case, valgrind was the fastest to instrument and easiest to run, as it required no compile-time library additions.

Incremental Persistent Index Statistics

This includes the number of indexed formulae, as well as the size of the persistent index, taken after every document is indexed. We built this to track the size of the index as more formulae are indexed. This provides accurate sizes for the persistent index, but can also be used as an scaled estimation for the temporary index.

These metrics were measured in 2 environments:

- MathWebSearch built using GNU-g++ toolchain on Linux
- MathWebSearch built using LLVM-clang toolchain on Mac

Next, we will present how each improvement incrementally impacted the size of the MathWebSearch index.

5.1.2 Narrow Identifiers

In figures 3 and 4 we present the index size for the baseline implementation, compared to the narrow identifiers implementation. In this test, narrow identifiers improve the index size from 304.42Mb to 279.59Mb for the GNU build and, respectively from 234.97Mb to 210.14Mb for the LLVM build. This is only 9%, respectively 11% reduction in memory footprint, lower that what we have predicted in section 4.2.1. We assume this is due to the fact that the library data structures (maps) in our index have higher memory costs than the size of the data we are providing. This overhead can be due to data padding, data structure internal implementation, as well as allocator metadata.

One can notice that the LLVM C++ standard template library has more memory efficient maps, its index being 33%, respectively 30% more efficient.

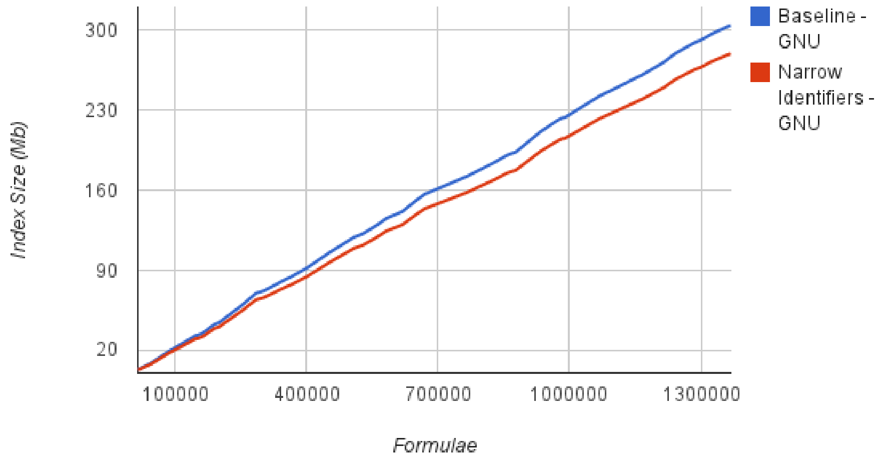


Figure 3: Narrow Identifiers optimization on GNU build

5.1.3 Space Optimized Map

In figures 5 and 6 we compare the size of an index with narrow identifier implementation with one which also has the optimized map implementation. Here, we see significant improvements in memory efficiency, as the index size drops from 279.59Mb to 118.50Mb (58% reduction) on GNU build, respectively from 210.14Mb to 118.52Mb (44% reduction). This confirms the fact that a large memory hog was the map implementation (especially in GNU standard library).

5.1.4 Persistent Index

In figures 7 and 8 we present the size of the index with optimized map implementation, compared to the persistent index implementation. Again,

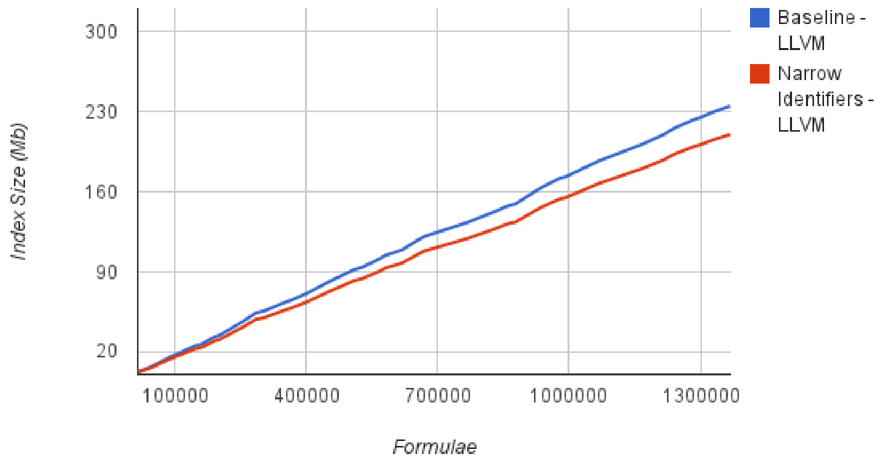


Figure 4: Narrow Identifiers optimization on LLVM build

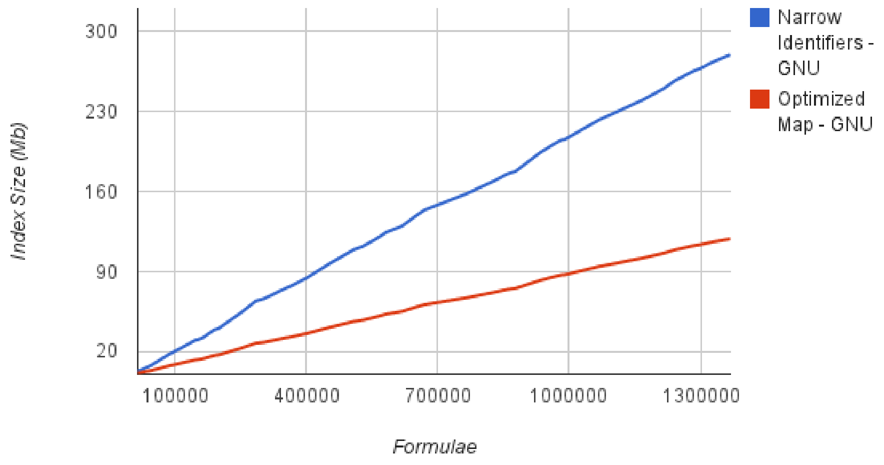


Figure 5: Map optimization on GNU build

we see significant improvements in memory efficiency, as the index size drops from 118.50Mb on GNU build, and, respectively 118.52Mb on LLVM build, to 57.90Mb (51% reduction). The persistent index has the same size on both platforms, as it is built though serialization, byte by byte.

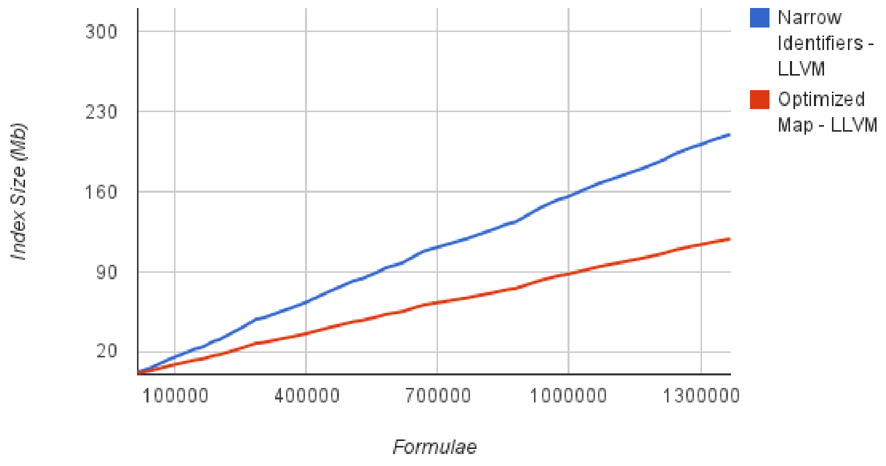


Figure 6: Map optimization on LLVM build

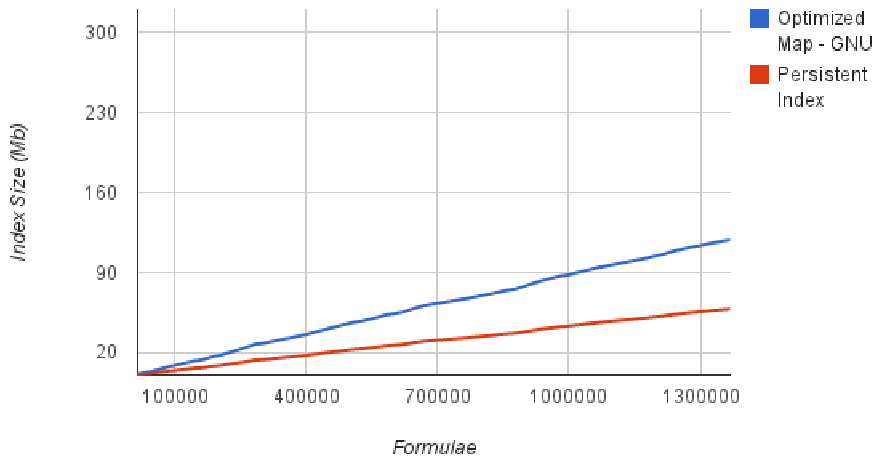


Figure 7: Persistent Index optimization on GNU build

5.1.5 Pointer Compression

In figure 9 we present the size of the persistent index, with and without pointer compression. We notice an improvement in memory efficiency, as the index size drops from 57.90Mb to 46.32Mb (20% reduction).

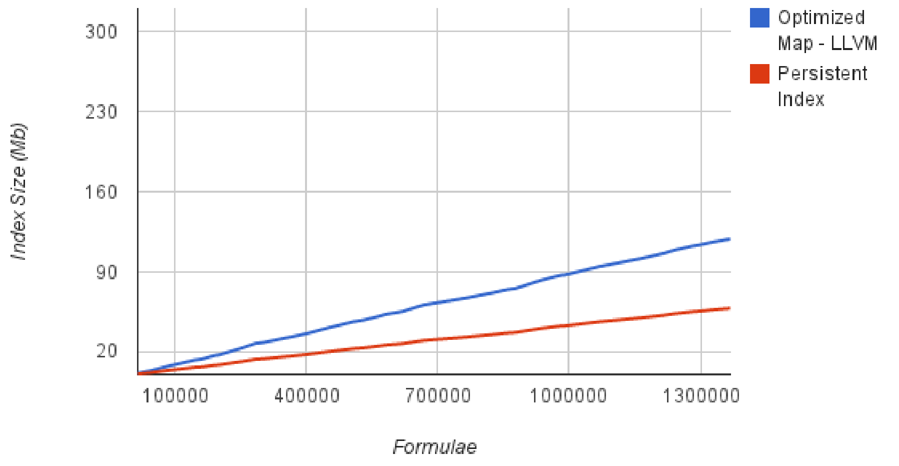


Figure 8: Persistent Index optimization on LLVM build

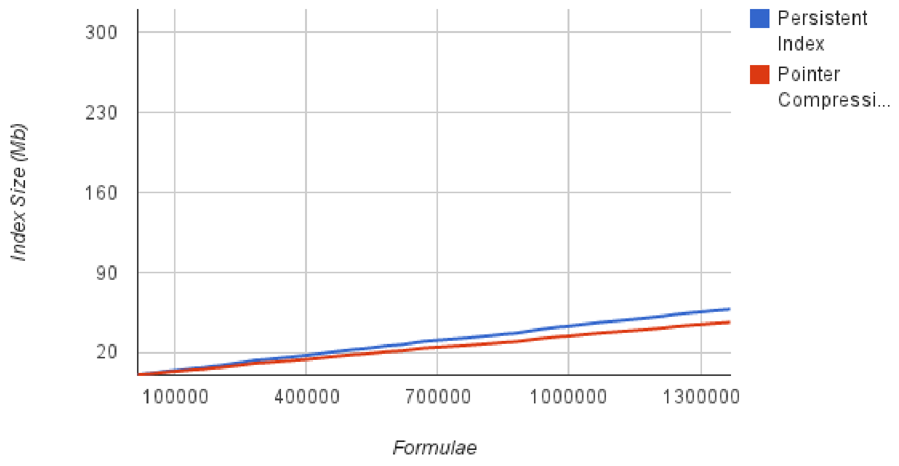


Figure 9: Pointer Compression optimization

5.1.6 Summary

Figure 10 presents an overview of all the optimizations. Overall, we have compressed the index from $304.42Mb$, respectively $234.96Mb$, down to $46.32Mb$. This is a compression of 85%, respectively 81%.

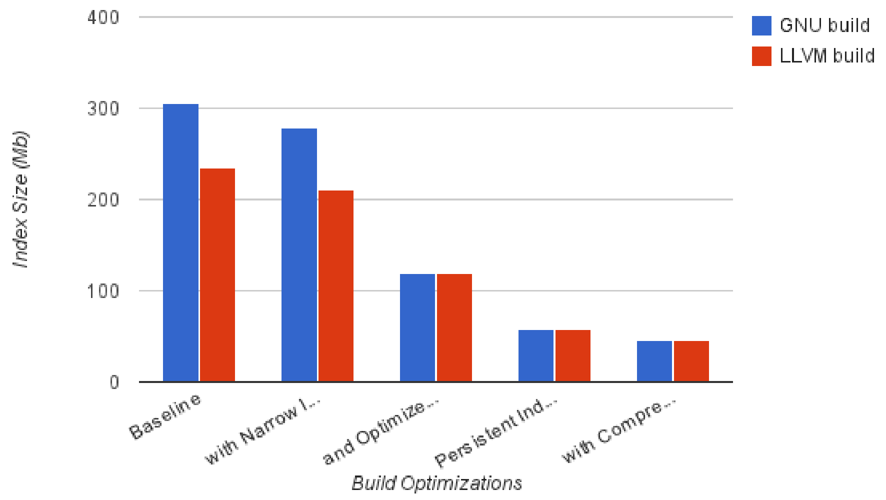


Figure 10: Pointer Compression optimization

Assuming linear index increase, if we index 500 documents in $46.32Mb$, an index of 1 million documents should be $90.47Gb$. Unfortunately, in a larger scale experiment we noticed that the size will be bigger. Using our latest implementation, we have indexed 105000 ArXiv documents, which were part of the NTCIR Challenge [28]. We generated a persistent index of 15.9Gb, which entails that an index of 1 million documents would require $151.43Gb$.

Note that the test dataset, the MathWebSearch build, as well as the raw evaluation data is available at [23].

5.2 Persistent Index Loading

As described in 4.1, one of our challenges was to mitigate the large indexing times which are inherent to large corpora. Here we will discuss how the persistent index solves this issue, with loading being significantly faster than indexing.

For the NTCIR challenge, we indexed 224Gb of harvests in 20h21m. By comparison, loading the resulting persistent index of 15.9Gb took 1m29s. Implementation-wise, index loading means memory mapping the index file into memory and validating the data by running CRC32 checksum on it.

To ensure our implementation is close to optimal, we ran the unix tool cksum, which computes the same CRC32 checksum, on the index file. This was done in 1m26s, only 4% faster than it takes for MathWebSearch to load the index.

5.3 NTCIR Challenge

This system has participated in the NTCIR Information Retrieval Challenge [28]. The task is to find documents matching a text and formula query.

The input dataset consists of paragraphs of 100,000 HTML documents from ArXiv. The query dataset has 50 queries submitted by mathematicians. Each query contains math formulae (as LaTeX, Presentation MathML and Content MathML) and keywords (as plain text). Results are submitted as lists of up to 1000 ranked hits (paragraphs) for each of the queries. These are judged against correct hits curated by mathematicians by the Text Retrieval Conference (TReC) evaluator [31].

Our system combines highly accurate formula search (MathWebSearch) with fuzzy text search. As a result, it returns two types of hits:

- high-score hits, which matched the formula and some of the keywords
- low-score hits, which matched only some of the keywords

High-score hits were scarce (under 10 per query) and we only found such hits for 30 queries. Low-score hits were more numerous (more than 100 per query) and were found for 49 queries.

Overall, we submitted 5260 hits for 49 out of 50 queries. Based on the TReC evaluation, our average precision was 24.03% and our average recall[6] was 34.68%.

6 Conclusions

We have presented the design and implementation of a system capable of harvesting, indexing and searching and analyzing text and formulae in (X)HTML documents. We have developed this with large corpora in

[6]The ratio between number of hits submitted and number of possible correct hits

mind, like the ArXiv corpus. As such, we have successfully tackled all challenges outlined in 4.1:

- the MathWebSearch in-memory formula index is 81% more memory efficient, as shown in 5.1.6
- the MathWebSearch index is now persistent and can be efficiently loaded, as shown in 5.2
- the system serves document snippets, with highlighted text and formulae, as shown in 4.9

Furthermore, we have implemented a formula analytics framework (4.6) and a formula purification module (4.8), as well as a tool which helps with change management in large corpora (4.7).

7 Applications and Future Work

The analytics framework opens up a number of possibilities for math analytics on large scale corpora. It can be used to find formulae with specific properties, quickly test hypotheses, or simulate standardization techniques. Some of these standardization techniques can be used to improve the quality of Content MathML generated by LaTeXML, thus benefiting all search systems which use LaTeXML-generated documents.

Another application for our work is seamless formula search and auto-completion integrated into math document editors. This would be a natural use-case for MathWebSearch, as tries are optimized for prefix search.

The user interface can be improved in several ways. Text and math input boxes can be merged, the engine being left to decide which tokens are math and which are text. This would make the interface more user friendly and avoid common mistakes like typing math or text in the wrong box. Another improvement we envision are search suggestions. For text input, we can use suggestions based on prefix queries in the English dictionary, while for math input we can use the Zentralblatt Glossary [16] (a human-curated list of mathematical entities).

References

[1] Akiko Aizawa, Michael Kohlhase, and Iadh Ounis. "Ntcir-10 Math Pilot Task Overview". In: *NTCIR Workshop 10 Meeting*. to appear. Tokyo, Japan, 2013, pp. 1–8. url: http://research.nii.ac.jp/ntcir/workshop/OnlineProceedings10/pdf/NTCIR/OVERVIEW/01-NTCIR10-OV-MATH-AizawaA.pdf.

[2] *AMD64 Online Developer Manual*. Aug. 17, 2014. url: http://developer.amd.com/wordpress/media/2012/10/24593_APM_v2.pdf (visited on 08/17/2014).

[3] Apache Software Foundation. *Lucene*. url: http://lucene.apache.org/ (visited on 05/04/2010).

[4] *arxiv.org e-Print archive*. url: http://www.arxiv.org (visited on 06/12/2012).

[5] *Bing Website*. Mar. 20, 2014. url: http://bing.com/ (visited on 03/20/2014).

[6] *C++ Standard Template Library Website*. Aug. 17, 2014. url: http://www.cplusplus.com/reference/stl/ (visited on 08/17/2014).

[7] *Clang Sanitizers Documentation*. Aug. 17, 2014. url: clang.llvm.org/docs/AddressSanitizer.html%20,%20clang.llvm.org/docs/MemorySanitizer.html,%20clang.llvm.org/docs/ThreadSanitizer.html (visited on 08/17/2014).

[8] *Clang-Format Documentation*. Aug. 17, 2014. url: http://clang.llvm.org/docs/ClangFormat.html (visited on 08/17/2014).

[9] *CMake Website*. Aug. 17, 2014. url: http://www.cmake.org/ (visited on 08/17/2014).

[10] *Elastic Search*. Feb. 20, 2014. url: http://www.elasticsearch.org/ (visited on 02/20/2014).

[11] *Git Website*. Aug. 17, 2014. url: http://git-scm.com/ (visited on 08/17/2014).

[12] *GitHub Website*. Aug. 17, 2014. url: https://github.com/ (visited on 08/17/2014).

[13] *GNU Debugger Website*. Aug. 17, 2014. url: http://www.gnu.org/software/gdb/ (visited on 08/17/2014).

[14] *Google Code Style Website*. Aug. 17, 2014. url: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml (visited on 08/17/2014).

[15] *Google Website*. Mar. 20, 2014. url: http://google.com/ (visited on 03/20/2014).

[16] Michael Kohlhase. *A Data Model and Encoding for a Semantic, Multilingual Glossary of Mathematics*. submitted to CICM 2014.

url: http://kwarc.info/kohlhase/submit/cicm14-smglom-datamdl.pdf.

[17] Michael Kohlhase, Bogdan A. Matican, and Corneliu C. Prodescu. “MathWebSearch 0.5 – Scaling an Open Formula Search Engine”. In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics (CICM) (Bremen, Germany, July 9–14, 2012). Ed. by Johan Jeuring et al. LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012, pp. 342–357. isbn: 978-3-642-31373-8. url: http://kwarc.info/kohlhase/papers/aisc12-mws.pdf.

[18] Michael Kohlhase and Corneliu Prodescu. “MathWebSearch at NTCIR-10”. In: *NTCIR Workshop 10 Meeting*. to appear. Tokyo, Japan, 2013, pp. 675–679. url: http://research.nii.ac.jp/ntcir/workshop/OnlineProceedings10/pdf/NTCIR/MATH/04-NTCIR10-MATH-KohlhaseM.pdf.

[19] Michael Kohlhase and Ioan Şucan. “A Search Engine for Mathematical Formulae”. In: *Proceedings of Artificial Intelligence and Symbolic Computation, AISC’2006*. Ed. by Tetsuo Ida, Jacques Calmet, and Dongming Wang. LNAI 4120. Springer Verlag, 2006, pp. 241–253. url: http://kwarc.info/kohlhase/papers/aisc06.pdf.

[20] Martin Líška, Petr Sojka, and Michal Ružicka. “Similarity Search for Mathematics: Masaryk University team at the NTCIR-10 Math Task”. In: *Proc. of the 10th NTCIR Conf. on Evaluation of Information Access Technologies*. 2013, pp. 686–691.

[21] *LLVM Debugger Website*. Aug. 17, 2014. url: http://lldb.llvm.org/ (visited on 08/17/2014).

[22] *Math Web Search*. url: https://trac.mathweb.org/MWS/ (visited on 01/08/2011).

[23] *MathWebSearch Evaluation Data*. Aug. 22, 2014. url: http://jupiter.eecs.jacobs-university.de/mws-eval/ (visited on 08/22/2014).

[24] Bruce Miller. *LaTeXML: A LaTeX to XML Converter*. url: http://dlmf.nist.gov/LaTeXML/.

[25] Jozef Misutka and Leo Galambos. “System Description: EgoMath2 As a Tool for Mathematical Searching on Wikipedia.org”. In: *Calculemus/MKM*. Ed. by James Davenport et al. LNAI 6824. Springer Verlag, 2011, pp. 307–309. isbn: 978-3-642-22672-4.

[26] *Node JS Package Manager Website*. Aug. 17, 2014. url: https://www.npmjs.org/ (visited on 08/17/2014).

[27] *Node JS Website*. Aug. 17, 2014. url: http://nodejs.org/ (visited on 08/17/2014).

[28] *NTCIR Homepage*. Aug. 22, 2014. url: http://ntcir-math.nii.ac.jp/ (visited on 08/22/2014).

[29] *NTCIR Workshop 10 Meeting*. to appear. Tokyo, Japan, 2013.

[30] *Phabricator Differential Website*. Aug. 17, 2014. url: http://phabricator.org/applications/differential/ (visited on 08/17/2014).

[31] *Text Retrieval Conference Homepage*. Aug. 22, 2014. url: http://trec.nist.gov/ (visited on 08/22/2014).

[32] *Valgrind Website*. Aug. 17, 2014. url: http://valgrind.org/ (visited on 08/17/2014).

[33] *Waffle.io Website*. Aug. 17, 2014. url: https://waffle.io/ (visited on 08/17/2014).

[34] *Web MIaS Demo Website*. Apr. 2, 2014. url: https://mir.fi.muni.cz/webmias-cicm-2014/ (visited on 04/02/2014).

[35] *XPath Reference*. 2010. url: http://www.w3.org/TR/xpath/ (visited on 06/05/2010).