SCALABLE SEARCHING AND RANKING FOR MELODIC PATTERN QUERIES

Philippe Rigaux
Cedric Lab, CNAM
`philippe.rigaux@cnam.fr`

Nicolas Travers
Leonard de Vinci, Research Center
Cedric Lab, CNAM
`nicolas.travers@devinci.fr`

ABSTRACT

We present the design and implementation of a scalable search engine for large Digital Score Libraries. It covers the core features expected from an information retrieval system. Music representation is pre-processed, simplified and normalized. Collections are searched for scores that match a melodic pattern, results are ranked on their similarity with the pattern, and matching fragments are finally identified on the fly. Moreover, all these features are designed to be integrated in a standard search engine and thus benefit from the horizontal scalability of such systems. Our method is fully implemented, and relies on ELASTICSEARCH for collection indexing. We describe its main components, report and study its performances.

1. INTRODUCTION

We consider the problem of searching large collections of digital scores encoded in a symbolic format, typically MusicXML [?], MEI [?, ?], or the forthcoming format of the W3C Music Notation Group [?]. These encodings are now mature and stable, and we can expect to witness in the near future the emergence of very large Digital Score Libraries (DSL). A representative example of such endeavors is the OpenScore initiative, which aims at publishing high-quality encoding of public domain sheet music. This potentially represents millions of scores, and gives rise to strong needs in terms of collection management tools tailored to the peculiarities of music representation.

In the present paper, we focus on the *content-based retrieval* problem. We consider the search mechanism where a user submits a monophonic *query pattern* in order to retrieve, from a very large collection of scores, those that contain one or more fragments "similar" to this pattern. We further require the search system to be *scalable*.

With this objective in mind, we propose two main contributions. First, we expose the design of the core modules of a search engine, namely pre-processing and data normalization, pattern-based search, ranking, and on-line identification of fragments that match the pattern query. Second, we propose a list of guidelines for integrating these modules in a standard information retrieval system, with two main benefits: reduction of implementation efforts, and horizontal scalability.

Our approach is summarized by Figure 1. Pre-processing, matching, occurrence extraction and ranking are standard steps in text-based information retrieval systems, adapted here to the specificities of music representation. Tokenization, stemming and lemmatization [?] are, in our case, replaced by a so-called *normalization* that simplifies the representation and improves the robustness of the result. The *matching step* operates on the normalized representation (query pattern and score content). Normalization and matching are detailed in Section 3.

Obtaining a full score in the result would be of little use if we were not able to identify all the fragments that actually match the pattern, called *pattern occurrences*. This is necessary, for instance, to highlight them in the user interface. The algorithm is described in Section 4.

Finally, the set of matching scores are sorted according to the similarity of their occurrences to the pattern. While pattern matching mostly relies on the melodic profile, the ranking method focuses on the rhythm (Section 5). Their combination produces results with highly relevant scores.

The rest of the paper (Section 6) covers our second contribution, namely the integration of our music retrieval components in a standard search engine. For the sake of concreteness, we detail this integration with ELASTICSEARCH (`https://elastic.co`).

We finally position our work with respect to the state of the art and lists some useful extensions that could enrich the search functionalities (Section 7).

2. PRELIMINARIES: SCORES, FRAGMENTS, AND PATTERNS

Given a *melodic pattern* P as input, the *pattern matching operation* retrieves all the *scores* such that at least one *fragment* matches P. Our approach focuses on the pitch and duration features, generally considered as the most important parameters for melodic similarity [?]. We model a score as a synchronization of *voices*, and each voice as a sequence of elements $< e_1, e_2, \cdots, e_n >$ with e_i in $\mathcal{E} \times \mathcal{D}$, where $\mathcal{E}$ is the domain of musical "events" (notes, chords, rest) and $\mathcal{D}$ the musical duration.

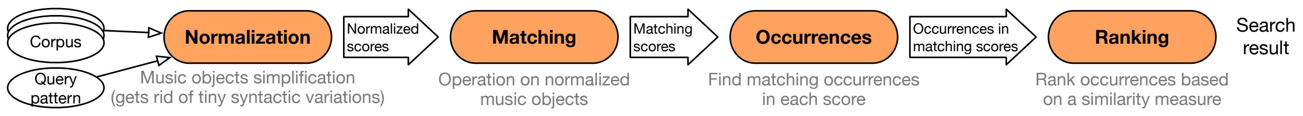

Corpus
Query pattern
Normalization
Normalized scores
Matching
Matching scores
Occurrences
Occurrences in matching scores
Ranking
Search result
Music objects simplification (gets rid of tiny syntactic variations)
Operation on normalized music objects
Find matching occurrences in each score
Rank occurrences based on a similarity measure

Figure 1. Overview of the main indexing and matching steps

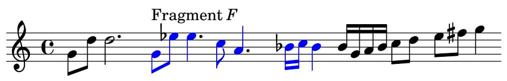


Figure 2. Voice V_I

Let us take the example of a voice V_I (Fig. 2). The blue fragment, denoted by F in the following, is used to illustrate the matching operation. V_I encodes a melody beginning with a G4 (semi-quarter), followed by a D5 (idem), a D5 (dotted half), etc. Using some pitches encoding mechanism, for instance the chromatic notation (number of semitones from the lowest possible sound), one obtains the representation of this voice as a sequence of pairs:
$V_I =< (34,8); (41,8); (41,3); (34,8); (42,8); (42,6); (39,8); \cdots >$

Given a voice V, we can derive other representations thanks to *transformation* functions. We consider two main categories of transformations: *simplifications* and *mutations*. They will be used in the normalization process.

Definition 1 (Simplifications) *A voice V can be transformed by the following simplification functions: $\epsilon(V)$, the sequence of pitches, $\pi(V)$ the sequence of pitch intervals between events in $\epsilon(V)$, and $\rho(V)$ the sequence of duration (without events) of V.*

Intuitively, $\epsilon(V)$ captures the melodic profile (sequence of note heights), $\pi(V)$ the relative evolution of pitches' heights, and $\rho(V)$ the rhythmic profile of a voice. Applied to V_I one obtains:

1. $\epsilon(V_I) =< 34, 41, 41, 34, 42, 42, 39, 36, \ldots >$
2. $\pi(V_I) =< 7, 0, -7, 8, 0, -3, -3, \ldots >$
3. $\rho(V_I) =< 8, 8, 3, 8, 8, 6, 8, 6, \ldots >$

Definition 2 (Mutations) *A mutation $M_{I_1 \to I_2}$ maps an interval I_1 to another interval I_2. A* mutation family *is a set of mutation functions.*

For example, $\mathcal{M}_D = \{M_{1\to2}, M_{3\to4}, M_{8\to9}\}$ denotes a subset of the family of *diatonic mutation* that transforms a minor second, third or sixth in, respectively, their major counterpart, and conversely.

Finally, a *fragment* is any subsequence of a voice. We will use the word "pattern" to denote the fragment supplied by some user as a search criteria.

3. NORMALIZATION AND MATCHING

The matching operation is a Boolean procedure that tells whether the pattern P and a fragment F are similar to one another. This similarity concept is subject to a trade-off between the precision (relevant part of the result) and the recall (global relevant scores over the result). This is a traditional information retrieval issue. Let us examine how it is translated in the realm of symbolic music representation.

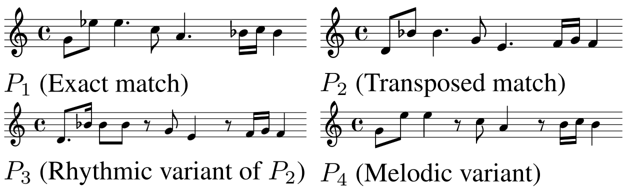


Figure 3. Several matching interpretations

3.1 Discussion

Fig. 3 shows several pattern variants, candidates to match with the fragment F of V_I (blue note heads in Fig. 2).

Exact Match. The strictest matching definition requires both the sequence of pitches (resp. $\epsilon(F)$ and $\epsilon(P)$) and the sequence of durations (resp. $\rho(F)$ and $\rho(P)$) to be identical. If we stick to this definition, F matches only with pattern P_1. The precision is then maximal but we will miss results that seem intuitive. F will not match for instance with the transposed pattern P_2, all other things being equal. This is probably too strict for most applications.

Transposed Match. Accepting transposition means that we ignore the absolute pitch and focus only on intervals, i.e., we compare $\pi()$ and $\rho()$, introducing flexibility in the melodic correspondence. In that case P_2 matches F.

Rhythmic Match. Next, consider pattern P_3 (Fig. 3), a rhythmic variant of P_2 that does not match F by exact rhythmic matching. Again, this definition seems too strict, since short rests, or slight duration adjustments, can typically be added or removed from a voice to denote a specific articulation, without severely affecting the music itself. P_3 matches F if we compare only $\pi(P_3)$ and $\pi(F)$, and ignore $\rho()$. Note that rhythmic changes involve not only rests and durations, but also repeated notes.

Melodic Match. Finally, P_4 is a pattern where intervals have been mutated. The initial minor sixth is replaced by a major sixth. Since such mutations can be found in imitative styles (e.g., counterpoint), it can make sense to accept them as part of the matching definition.

How far are we ready to go in the transformation process? Fig. 4 shows two extreme examples. Pattern P_5 matches F with respect to the sequence of intervals ($\pi(P_5) = \pi(F)$), whereas pattern P_6 is a rhythmic match ($\rho(P_5) = \rho(F)$). It seems clear that these patterns are quite far from the considered fragments and that, at the very least, they should not be given the same importance in the result set than the previous ones.

Music similarity has been studied for decades now. It seems obvious that there is no ideal solution that would suit all situations [**?**, **?**] since similarity judgments depend on many aspects. However, our goal here is *not* to compute all the similarities, but to provide a filtering mechanism that gets rid of the scores that do not match the query pattern.

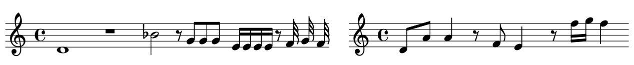

P_5 (Melodic matching) P_6 (Rhythmic matching)

Figure 4. Two examples that match F on one dimension

Algorithm 1 Voice normalization
```
1: procedure VNORM($V$)
2: Input: A voice $V$
3: Output: A voice $V'$, normalization of $V$
4:     $V' \leftarrow V$
5:     Normalize all note durations in $V'$ to a quarter.
6:     Merge repeated notes from $V'$.
7:     Remove rests from $V'$
8:     return $V'$
```

This mechanism should be simple, efficient, and plugable in a standard search engine. It does not require to apply a costly similarity function to the whole collection.

In this perspective, matching-based retrieval is a first step operated to filter out a large part of the collection. A simplified similarity function can be used to top rank relevant scores. It is then easy to develop further investigations (*e.g.,* specialized similarity function) on the result set.

3.2 Normalization

It is generally considered that rhythm plays a prominent role in the perception of similarity. We therefore rank the result according to the rhythmic likeness of each retrieved fragment with the pattern. Filtering is based on the melodic profile, and its impact depends on how we simplify this profile in the normalization step. Two extreme choices are either to keep the exact sequence, increasing the precision, or to extract the melodic contour, increasing the recall.

In information retrieval systems, normalization is part of pre-processing steps, usually called *analyzers* and can be tuned by the administrator. This flexibility should be adopted for the symbolic music retrieval as well.

Our current implementation relies on the VNORM() algorithm (Alg. 1), applied to both the pattern and each voice in the corpus. Fig. 5 shows the voice normalization applied on patterns F, P_1 (top) and P_2, P_3, P_5 (bottom). In both cases the sequence of intervals obtained by $\pi()$ on the normalization is <6,-3,-3,1,2,-2>.

3.3 Matching

Definition 3 *A score S* matches *a pattern P iff, for* at least *a voice v in S, and* at least *a pair $[b, e], e > b$ of offsets (positions) in v, where the set of voice fragments $v[b] \cdots v[e]$ that match P are called the* matching occurrences *of S.*

$$\pi(\text{VNORM}(P)) = \pi(\text{VNORM}(v[b] \cdots v[e]))$$

4. FINDING MATCHING OCCURRENCES

Once matching scores have been extracted from the repository, it is necessary to identify the corresponding sequences of pitches that match the given query pattern (on normalized *n*-grams). For this, we need to look forward to exact match between intervals in the query pattern, and pitches in score's voices. Algorithm 2 produces a list of matching pitches which will be ranked in the next section.

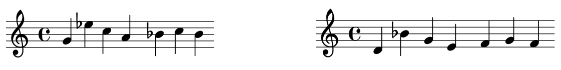

F and P_1 P_2, P_3, P_5

Figure 5. Voice normalization.

Algorithm 2 Finding matching occurrences
```
1: procedure FINDINGOCCURRENCES($V$, $Q$)
2: Input: A voice $V$, a query $Q$ of intervals
3: Output: A set $L$ of fragments
4:     for $p$ in $LCS(V, Q)$ do          ▷ List of matching patterns
5:         $L \leftarrow L \cup p$
```

This procedure processes a voice V with a given query pattern Q. The LCS [**?**, **?**] algorithm (*Longest Common Subsequence*) will give in output each matching pattern in V. It will verify if the pattern Q matches any interval between two successing pitches from V. The LCS algorithm iterates on each pitch of V to check each eligible subsequence, especially for matching patterns contained into repeating subfragments in Q.

5. RANKING

Given a set of fragments that match a pattern P, we now want to sort them according to a similarity measure, and put on top the ones that are closest to P. Assume that the search pattern is our previous F (Fig. 2, blue heads) and that the result set is $\{P_1, P_2, P_3, P_5\}$. For all, function $\pi()$ composed with the normalization VNORM yields <6,-3,-3,1,2,-2>. Intuitively P_1 and P_2 should be ranked first, and P_5 should be ranked last.

It is important to note that this ranking applies to fragments that have an identical melodic pattern. We take advantage of this specificity to operate at two levels. The first level measures the similarity of the "melodic rhythm", i.e., the respective duration of pairwise intervals in each fragment. To this end we define the notion of *blocks*.

Definition 4 *Let F be a fragment such that $\pi(\text{VNORM}(F)) = < I_1, \cdots, I_n >$. By definition of π and* VNORM, *each interval $I_j, j \in [1, n]$ is represented in F by a sequence $< p^i_1, e^i_2, \cdots, e^i_{k-1}, p^i_k >$ such that:*

- *p^i_1 and p^i_k are two distinct pitches, and $interval(p^i_1, p^i_k) = I_i$*
- *each $e^i_l, l \in [2, k-1]$ is either a rest, or a pitch such that $e^i_l = p^i_1$*

We call $< p^i_1, e^i_2, \cdots, e^i_{k-1} >$ the block *B_i of I_i in F.*

A block is the largest subsequence of a fragment that covers a non-null interval. The concept of block is illustrated by Fig. 6 for P_1, P_3 and P_5.

The first level of the ranking function evaluates the similarity of two fragments F_1 and F_2 by comparing the blocks

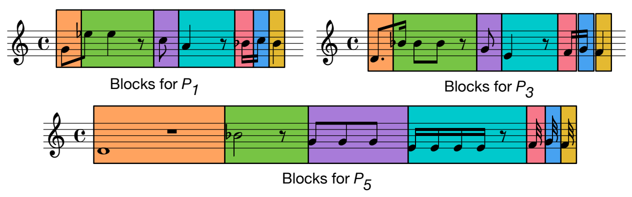


Figure 6. Blocks for P_1, P_3, and P_5

Algorithm 3 Ranking procedure

```
1: procedure RANKING($F_1, F_2$)
2: Input: $F_1$, $F_2$, such that $\pi(\text{VNORM}(F_1)) = \pi(\text{VNORM}(F_2))$
3: Output: a similarity $s \in [0, 1]$
4:     $s \leftarrow 0; d_1 \leftarrow duration(F_1); d_2 \leftarrow duration(F_2)$
5:     for $i := 0$ to $n$ do                         ▷ Loop on the blocks
6:         $s \leftarrow s + |dur(B_i^1)/d_1 - dur(B_i^2)/d_2|$
7:     if $s = 0$ then $s \leftarrow TieBreaking(F_1, F_2)$
8:     return $s/2$                          ▷ Euclidian normalization [?]
```

pairwise durations. The rationale is that if these durations are similar, the only difference lies in either repeated notes or rests inside each block. Fig. 6 shows for instance that block durations in P_1 and P_3 are exactly the same, which makes them almost identical. The difference is internal to each block (for instance block 2, in green). On the other hand, P_1 and P_5 turn out to be quite dissimilar.

The RANKING function is simple and efficient (Algorithm 3). We first normalize the fragment duration, and sum up the difference of durations between pairs of blocks.

If it turns out that all block durations are pairwise identical, a tie-breaking function has to be called. This is the only situation where we might have to examine internal of blocks. By definition of blocks, this internal representation only consists of rhythmic data: rests and repeated notes. Any standard text comparison method (edit distance, Levhenstein distance) can be used.

6. INDEXING

We now describe how our functions can be integrated in a search engine. For the sake of concreteness, our description relies on ELASTICSEARCH, but the method works for any similar system (e.g., Solr) that uses inverted index.

6.1 Encoding

An index in ELASTICSEARCH is built on JSON documents. Each field in such a document can be either *indexed*, *stored* or both. Indexing a field means that ELASTICSEARCH supports full-text searches on the field's content. Storing a field means that the field's content is stored in the index. Our index features an n-gram field for searching, and a `sequence` field for the ranking.

Given a voice V and the sequence of intervals of its normalization $\pi(\text{VNORM}(V)) =< I_1, \cdots, I_k >$, we compute the list of n-grams $\{<I_i, \cdots, I_{i+n-1}>, i \in [1, k-n+1]\}$, where n, the n-gram size, is an index configuration parameter. If, for instance, the sequence of intervals is <6,-3,-3,1,2,-2>, the list of 3-grams is {<6,-3,-3>, <-3,-3,1>, <-3,1,2>, <1,2,-2>}.

Each n-gram is then encoded as a character string which constitutes a *token*. These tokens are finally concatenated in a large character string, separated by a white space. Positive integers are encoded with `a`, `b`, `c`, etc., and the minus sign by `m`, as illustrated in the following:

```
{"query": {"match_phrase":
  {"ngram": "fmcmc mcmca mcab abmb"} } }
```

6.2 Searching

We can then run keyword queries and, more importantly, *phrase queries* where ELASTICSEARCH retrieves the fields that contain a list of tokens that appear in a specific order. The previous query shows the"*match_phrase*" query which searches the n-gram sequence.

The search engine then does the rest of the job for us. It finds all the indexed documents such that the n-gram field contains the phrase. However, by default, ranking is based on textual features that do not match what we expect. We therefore need to replace the default ranking method.

6.3 Ranking

Ranking functions can be overridden in ELASTICSEARCH [?, ?]. To this end, we must provide a Java function that implements the ranking method exposed in Section 5. This function is called at query time and produces a similarity score for each voice. The result is sorted on this value, and made accessible to the client application via an iterator-like mechanism [?] called *SearchScript*.

Our ranking function operates on a voice to identify the matching occurrences, and to measure the similarity between the search pattern and each occurrence. We must store in ELASTICSEARCH an encoding of the voice that can be accessed during the query evaluation.

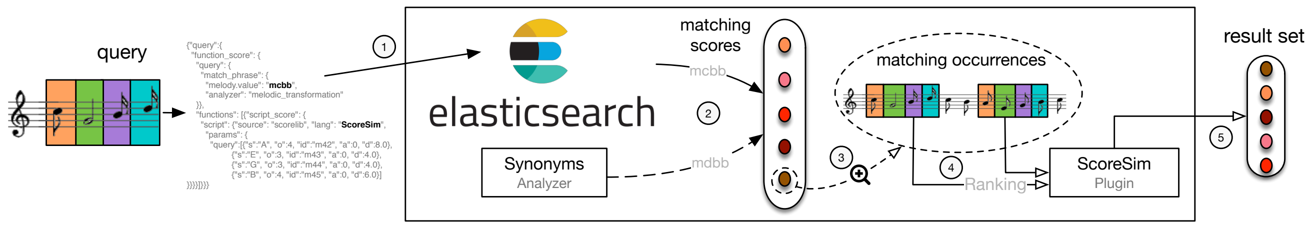
6.4 Query expansion

Our method relies on a strict matching of a sequence of intervals. Since accepting unbounded melodic transformations would likely return the whole database, we can consider those that can be seen as meaningful from a musical point of view. For instance, diatonic mutations of intervals (e.g., accepting both minor and major thirds or sixths in the matching operation) probably makes senses and can improve significantly the recall of our method.

We have integrated the synonym query expansion feature of ELASTICSEARCH engine (e.g., the ability to map “car” to “vehicle”) to implement this feature. To achieve this, we decided to produce a list of synonyms for major thirds or sixths. These means that an interval 'c' can be similar to 'd' or interval 'h' to 'i'. The following n-grams are then considered to be: similar`cbh, dbh, dbi, cbi`

In order to integrate the list of synonyms to ELASTICSEARCH, it is given to the index as an analyzer *"melodic_transformation"* (illustrated in the query Section 6.5) and matching n-grams are merged at query time.

To find the matching occurrences, we need to modify Algorithm 2 in order to integrate the synonyms where intervals (in the `LCS` algorithm) for major third and sixth



```
{"query":{
  "function_score": {
    "query": {
      "match_phrase": {
        "melody.value": "mcbb",
        "analyzer": "melodic_transformation"
    }},
    "functions": [{"script_score": {
      "script": {"source": "scorelib", "lang": "ScoreSim",
        "params": {
          "query":[{"s":"A", "o":4, "id":"m42", "a":0, "d":8.0},
                   {"s":"E", "o":3, "id":"m43", "a":0, "d":4.0},
                   {"s":"G", "o":3, "id":"m44", "a":0, "d":4.0},
                   {"s":"B", "o":4, "id":"m45", "a":0, "d":6.0}]
}}}}]}}}
```

Figure 7. `ScoreSim` integration in ELASTICSEARCH

can be authorized, then: 1.5 ∼ 2 and 3 ∼ 3.5. ELASTICSEARCH can be further enhanced by taking into account a similarity measure between those synonyms (*e.g.*, oriented graph weighted with similarity values between synonyms).

6.5 Implementation

The integration of our approach in ELASTICSEARCH needs to preprocess musical scores to normalize them, and to compute the ranking in the search engine on query-time.

The first step consists in a Python scripts that normalizes voices from scores (Alg. 1), extracts *n*-grams, and produce JSON documents sent to the ELASTICSEARCH REST API. Documents also contain corpus and opus ids, syllables from lyrics voices which can be queried.

Second, to take into account synonyms in ELASTICSEARCH, the list of *n*-gram synonyms needs to be set offline. We have generated the list of all combinations of third and sixth transformation for each possible *n*-gram available in the repository. This list is imported in ELASTICSEARCH and then processed on-the-fly for each query.

The third step integrates the `ScoreSim` scoring module as a Java program in ELASTICSEARCH. For this, a *SearchScript* needs to be inherited in order to produce a plugin for ELASTICSEARCH. This plugin takes queries' parameters and instantiates a scoring function which will process every matching scores. The scoring function `ScoreSim` implements Algorithms 2 and 3.

The query below integrates all features that are proposed in our approach: *n*-gram search (*melody.value*), synonyms analyzer (*melodic_transformation*), and the `ScoreSim` function (*script_score*). In the latter, the parameter "*query*" gives the list of pitches used in order to produce the score value from Algorithm 3. The query params gives the sequence of the music items (octave, pitch and duration) that constitutes the pattern in order to provide distances and rank the result set.

```
{"query":{ "function_score": {
    "query": { "match_phrase": { "melody.value": "mcbb",
                   "analyzer": "melodic_transformation"}},
    "functions": [{"script_score": {
      "script": {"source": "scorelib", "lang": "ScoreSim",
        "params": {"query":[
            {"s":"A", "o":4, "id":"m42", "a":0, "d":8.0},
            {"s":"E", "o":3, "id":"m43", "a":0, "d":4.0},
            {"s":"G", "o":3, "id":"m44", "a":0, "d":4.0},
            {"s":"B", "o":4, "id":"m45", "a":0, "d":6.0}]}}}}]}}}
```

Figure 7 shows the querying process with the following steps: **1)** transform the melodic pattern into the ELASTICSEARCH DSL (Domain Specific Language), **2)** ELASTICSEARCH gets all the matching score corresponding to the given *n*-gram and eventually to their synonyms, **3)** instantiate the `ScoreSim` plugin and process every score, **4)** extract occurrences on each instance and then its score value, **5)** and finally, ELASTICSEARCH sorts the whole result-set according to the produced scores and sends the result.

	q1	q2	q3	q4
Querying pattern	demc	mcmcmc	bmbb	bmbmc
Nb of matching scores	204	719	877	2,225
Nb with query expansion	242	1867	877	2,236

Table 1. Query patterns

6.6 Performance

In order to study the impact of our approach on the computation time, we apply different queries on our corpora. The corpora size varies by cumulating several corpus, representing up to 4,950 polyphonic scores. It is composed of the whole corpora available here [1] composed of *Francœur, Méthodes, Motet, Psautiers, Sequentia, Timbres*, etc.

To study the effect of the matching process, we have chosen four different queries to apply with various patterns, from infrequent to more frequent ones, based on the popularity of the stored *n*-grams. Table 1 gives for each query pattern the corresponding total number of matches in the corpora and the number after applying the query expansion (synonyms). We can see that query q2 is clearly expanded since `mcmcmc` has 7 synonyms and produces a large number of matches (2.5 times more). At the opposite, query q3 has no synonyms and do not enlarge its result-set.

Figure 8 shows the evolution of the number of matching scores wrt the corpus size. It gives both matching scores for normal (plain lines) and expanded queries (dashed lines). According to the synonyms, we can see that q1 and q4 provide few more matchings, while q2 witnesses a different behavior where the number of matchings grows proportionally with the number of synonyms.

The execution time is plotted in Figure 9 for the 4 different queries. It shows both normal pattern queries (plain lines) and expanded queries (dashed lines). This allows to investigate both the robustness with respect to various results sizes, and issues related to false positives.

Each query is sub-linear in the result size. Query q4 is a frequent pattern which returns 2,225 matching scores (45% of the corpora). It is executed in 277 ms. The small number of synonyms has few impacts on the global processing time. At the opposite, q1 is extremely efficient due to its

[1] NEUMA repertory: `http://neuma.huma-num.fr/home/`

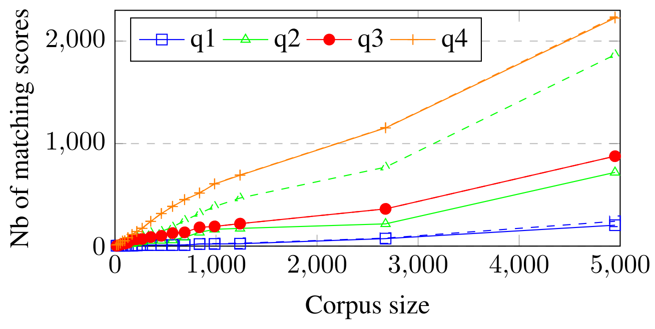


Figure 8. Nb of matching with and without synonyms

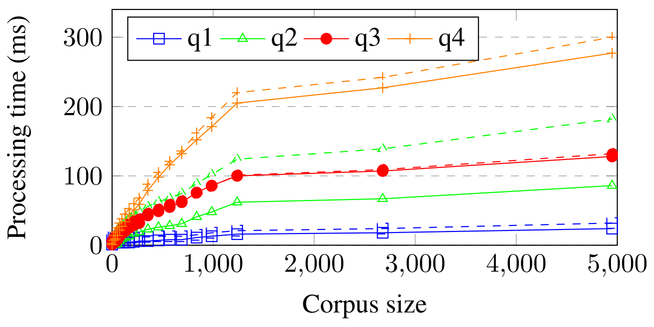


Figure 9. Execution time with and without synonyms

selectivity. It produces 204 scores in 22 ms. One interesting effect can be seen for query q2 where the number of matching synonyms leads to more computation time but it only 2.1 times more (for 2.5 times more matching scores).

Those experiments show that the ranking function takes less than 0.12 ms to process each score on a single ELASTICSEARCH node (server). The corpus can be spread on several nodes in order to scale it up horizontally.

7. RELATED WORK

Our approach combines similarity searches based on textual music encoding, scalable search, and rhythm-based ranking. The novelty of this approach is their association in a consistent setting. The rather trivial implementation makes, in our opinion, our solution quite attractive.

Music similarity has been an active MIR research topic over the last decades [**?**]. The general goal is to evaluate the likeness of two musical sequences. A major problem raised by this definition is that similarity judgments are highly dependent on both the content being compared and on the user taste, culture, and experience [**?**, **?**]. A recent survey [**?**] summarizes the recent trends observed in the SMS track of the MIREX competition.

A similarity method is characterized by the choice of musical parameters. Pitches and durations are generally considered as expressive enough. Using sequences to represent both parameters is primarily motivated by our objective to integrate our methods in a standard search engine, and to benefit from an index structure. Some important parameters, e.g., metric accent, structure or harmonic are ignored because they often lead to tree-based encodings that are hardly indexable. Geometric approaches, such as [**?**], are also less suitable in this indexing perspective. Multidimensional structures are complex, and their performances are known to fall down as the dimension increases [**?**], and not yet integrated to off-the-shell search engines.

Textual encoding of symbolic music representation is an attractive idea in order to use text algorithms. The *HumDrum* toolkit [**?**] relies on a specialized text format and adapts Unix file inspection tools for music analysis. Exact and approximate string matching algorithms for melody matching have been used in *ThemeFinder* [**?**, **?**] or *Musipedia* [**?**]. Many algorithms for efficient computation of similarity matching through exhaustive searches have been proposed [**?**, **?**, **?**, **?**]. Specialized rhythm similarity functions are proposed and compared in [**?**].

Text-based approaches are simple solutions, with two important limitations. First, combining pitches and rhythm in a single character string for instance is not easy, and small music variants may result in important syntactic differences. Second, these methods do not scale since the whole database has to be inspected for each query. Several indexing methods have been suggested for the edit distance [**?**, **?**]. The *Dynamic Time Warping* distance is another popular method, for which sophisticated indexing structures have been proposed [**?**,**?**]. None of them is available beyond research prototypes.

The easiest way to benefit from an inverted index is to split musical sequences in n-grams. This has been experimented in several earlier proposals [**?**,**?**,**?**,**?**]. Each n-gram plays the role of a “token” and search methods apply.

Ranking is an essential part of an information retrieval system. We believe that our proposal, which combines **1)** a pre-processing normalization step, **2)** a melodic profile search and **3)** a rhythmic profile ranking, completes earlier attempts to adapt text-based retrieval to music retrieval, and results in a complete workflow which achieves a satisfying trade-off between the filtering impact, the ranking relevancy and the overall efficiency.

8. CONCLUSION

We described in this paper a practical approach to the problem of indexing pattern-based searches in a large score library. Our solution fulfills three major requirements for an information retrieval system: (i) it supports search with a significant part of flexibility, (ii) it proposes a ranking method consistent with the matching definition, and (iii) it brings scalability thanks to its compatibility with the features of state-of-the-art search engines. We fully implemented our solution, including the internal ranking function for ELASTICSEARCH, and we will be pleased to supply our software components to any interested institution that wishes to propose a content-based search mechanism.

Score analyzers extension. Alg. 1 with the normalization of voices can be extended with ad hoc music *analyzers*: management of grace notes, the simplification of melodic profiles, treatment of repeated notes, or cross-voice melodies, to name a few.

Acknowledgments. This work is funded by the French ANR project MUNIR

9. REFERENCES