# Scalable Searching and Ranking for Melodic Pattern Queries

(In the initial submission, omit all the following author information to ensure anonymity during peer review.)

Firstname Lastname
Sound Computing Group
University of Anywhere
1234 Anywhere Street
Anywhere, Anwhere 012345 USA
email@email.com

**« AUTHOR TELEPHONE (not for publication): +44 999 999 9999 »**

# Scalable Searching and Ranking for Melodic Pattern Queries

March 19, 2019

**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. All the components are released in open source on Github for the community of people and institutions managing large collections of digitized scores.

## 1 Introduction

We consider the problem of searching large collections of digital scores encoded in a symbolic format, typically MusicXML [Good (2001)], sometimes MEI [Rolland (2002); MEI (2015)], or the forthcoming format proposed by the W3C Music Notation Group [MNX (2018)]. 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 (http://openscore.cc), 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 common 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 several fragments "similar" to this pattern. We further require the search system to be *scalable*, i.e., it should be able to cope with very large DSL containing millions of scores with instant response time. Our assumptions on scores, fragments, and patterns, as well as scalability requirements, are developped in the first section of the paper.

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 system, adapted here to the specificities of music representation. Tokenization, stemming and lemmatization [Manning et al. (2008)] are, in our case, replaced by a so-called *normalization* that simplifies the representation of

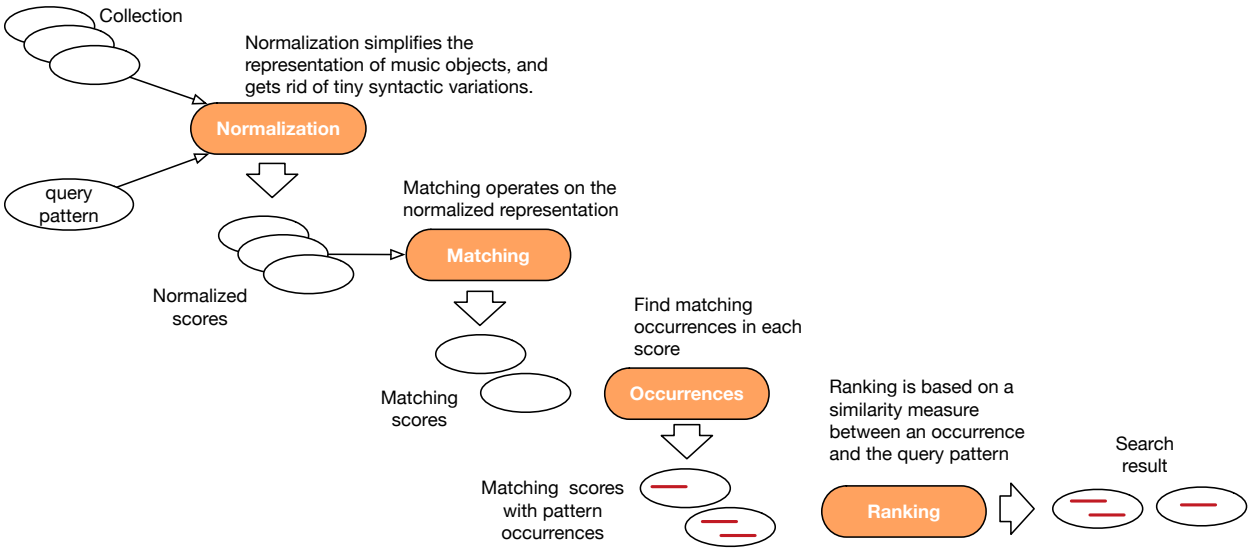



*Figure 1. Overview of the main indexing and matching steps*

music objects and improves the robustness of the result. The *matching step* then simply operates on the normalized representation of both the query pattern and the scores content. Normalization and matching are detailed in Section 3.

Obtaining a full score in the result of a pattern-based query 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. Occurrences identification operates on the full score representation. 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. Their combination produces results where highly relevant scores regarding both criteria are top-ranked. The ranking method is presented in Section 5.

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), however the method is actually applicable to any system relying on standard inverted index structures.

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).

The paper is deliberately practice-oriented. Anyone wishing to equip a score library with a search mechanism should be able to do so quite easily by following our methodology. To this end, we also created a GitHub repository where our components are freely available.

## 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 $P$. The concepts of scores, fragments, patterns and matching are defined in turn, and illustrated by the example of Figure 2 that shows the initial measures of a typical music score.

Our approach focuses on the pitch and duration features, generally considered as the most important parameters for melodic similarity [prince (2014)]. 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

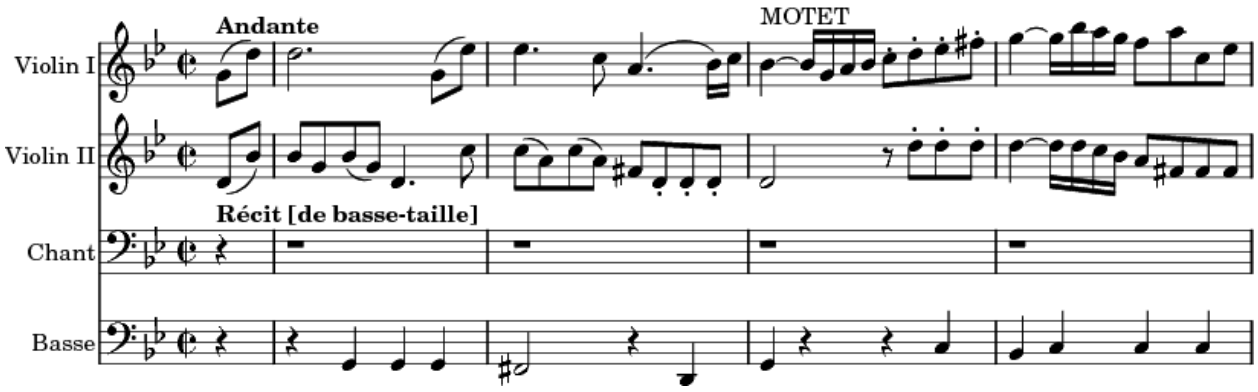



*Figure 2. Excerpt of a polyphonic score*

the domain of musical "events" (notes, chords, rest) and $\mathcal{D}$ the musical duration.

**Example 1.** *The musical score of Figure 2 consists of four voices (labeled as 'Violin I', 'Violin II', 'Chant' and 'Basse'), and each voice is a sequence of musical events.*

*Voice $V_I$ ('Violin 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 semi-tones from the lowest possible sound), one obtains the representation of this voice as a sequence of pairs:*

$$V_I =< (34, 8); (41, 8); (41, 3); (34, 8); (42, 8); (42, 6); (39, 8); \cdots >$$

Figure 3 shows the content of voice "Violin I" (note how many notation elements have been removed). The blue fragment, denoted by $F$ in the following, is used to illustrate the matching operation in the following.

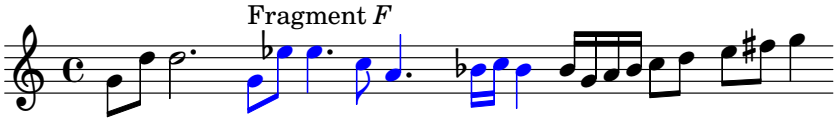



*Figure 3. Voice $V_I$ (Violin I of Figure 2)*

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 as part of 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.

**Example 2.** *Applying the simplification functions to $V_I$ yields the following results:*

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. $\rho(V_I) =< 8, 8, 3, 8, 8, 6, 8, 6, \ldots >$

A voice can also be transformed by applying mutations to the sequence of intervals of its events.

**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.*

We will denote for instance as $\mathcal{M}_D = \{M_{1\to2}, M_{3\to4}, M_{8\to9}\}$ a subset of the family of *diatonic mutation* that transforms a minor second, third or sixth in, respectively, their major counterpart, and conversely. If we mute the 4th and 6th intervals in $V_I$ with $\mathcal{M}_D$ one obtains Fig. 4.

---

**Nicolas**: *4° et 6°? pas plutot 1, 4 et 8?*

---

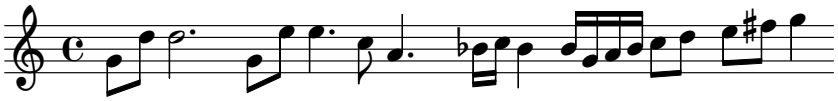


*Figure 4. Voice $V_I$ transformed with diatonic mutations*

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. Technically, there is no distinction between voices and fragments, apart from the context where they are used.

# 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. The definition of this similarity concept is subject to a trade-off between the precision (measuring the part of the result that is indeed relevant to the search) and the recall (measuring how many of the relevant scores are part of the result). This is a traditional information retrieval issue. Let us examine how it is translated in the realm of symbolic music representation.

## 3.1 Discussion

Fig. 5 shows several pattern variants, all candidates to match with the fragment $F$ of $V_I$ illustrated on Fig. 3 (blue note heads).

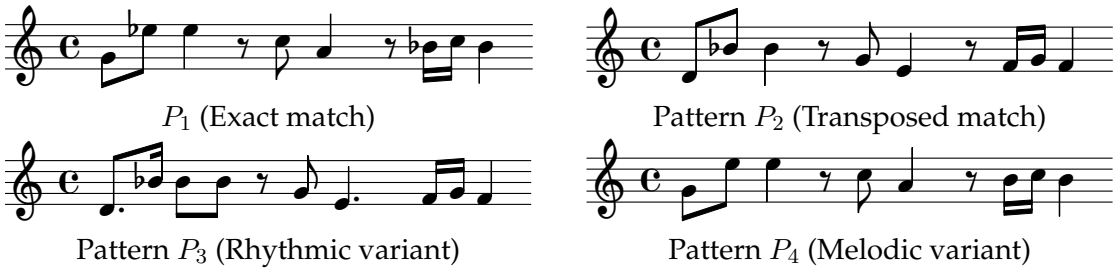



*Figure 5. Several matching interpretations*

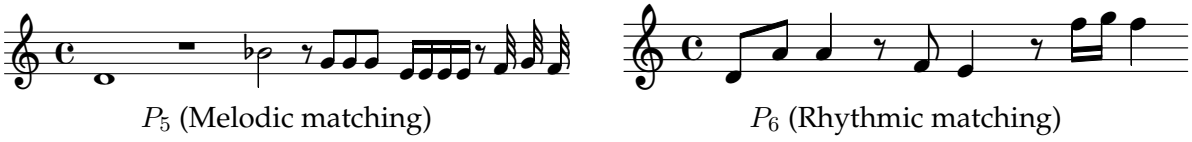
**Exact Match** The strictest matching definition requires both the sequence of pitches (resp. $\epsilon(F)$ and $\epsilon(P)$) and the sequence of durations (resp. $\rho(F)$ and $\rho(P)$) to be identical. If we stick to this definition, $F$ matches only with pattern $P_1$. The precision is then maximal (all matched fragments are relevant) 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 $\rho()$. This introduces some flexibility in the melodic correspondence. In that case $P_2$ matches $F$.

**Rhythmic Match** Next, consider pattern $P_3$ (Fig. 5), a rhythmic variant of $P_2$. Although quite close to $P_2$, it does not match $F$ if we require 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), and 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. 6 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 shoud not be given the same importance in the result set than the ones previous ones.



$P_5$ (Melodic matching) $P_6$ (Rhythmic matching)

*Figure 6. Two examples that match F on one dimension*

Music similarity has been studied for decades now. It seems obvious that there is no ideal solution that would suit all situations since similarity judgements depend on many aspects. Jones et al. (2007); Ens et al. (2017) However, our goal here is *not* to compute the full, complete and precise list of matching scores, but to provide a filtering mechanism that gets rid of most of the scores that do not match the query pattern. This mechanism should be simple, efficient, and plugable in a standard search engine. This avoids applying a costly similarity function to the whole collection.

In this perspective, matching-based retrieval is a first step operated for performance reasons. Scores in the result set delivered by this first step might be subject to further investigations if needed, for instance by applying a specialized similarity function. This led us to adopt the following design guidelines:

1. The result set should be ordered in such a way that the most relevant scores are top-ranked.
2. The filtering must be effective enough to avoid scanning a large part of the collection for each query.
3. The trade-off between recall and precision is tuned by a normalization applied during a pre-processing step.

### 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 (first criterion). Filtering is based on the melodic profile, and its impact depends on how we simplify this profile in the normalization step. Two extremes choices are either to keep the exact sequence of pitches, increasing the precision, or to extract the melodic contour, increasing the recall.

In information retrieval systems, normalization is part of a sequence 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 (Algorithm 1), applied to both the pattern and each voice in the collection of scores. We defer a more general discussion on the normalization step to the concluding remarks.

**Algorithm 1** Voice normalization

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

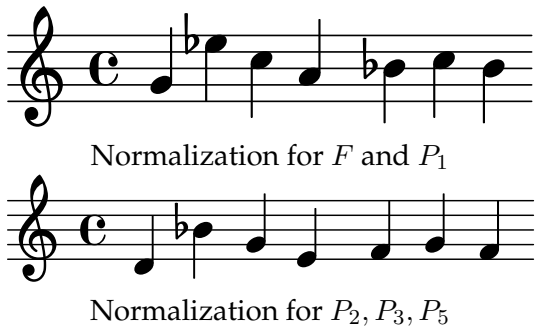



*Figure 7. Voice normalization.*

Fig. 7 shows the voice normalization operated for 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

The matching operation is defined as follows.

**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 $v$,*

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

*The set of voice fragments $v[b] \cdots v[e]$ that match $P$ are called the* matching occurrences *of $S$.*

# 4 Finding matching occurrences

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

**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:  Variables: F a list of pitches
5:              q ← first(Q)
6:      for p in V do                                          ▷ Loop on the pitchs
7:          if q == last(Q) then                               ▷ F fully matches Q
8:              L ← L ∪ F
9:              F ← ∅
10:             q ← first(Q)
11:         else                                               ▷ A new interval
12:             q ← following(q)
13:             if interval(q, preceding(q)) != interval(p, preceding(p)) then
14:                 F ← ∅                                      ▷ Mismatching interval
15:                 q ← first(P)
16:         F ← F ∪ p
```

**Nicolas**: *Attention aux motifs contenant sous-motif au début - regex / LCCS*

This procedure processes a voice $V$ with a given query pattern $Q$. For each pitch $p$ from $V$, it verifies first if the interval between $p$ and its previous pitch (line 7) is not null, it means that we have to check the following query block. The `interval` function returns 0 when a pitch $p$ has no preceding pitch (first pitch of $V$).

If $q$ is the last pitch of $Q$ (line 8), $F$ is then a matching occurrence that has to be added in output (line 9-11). Otherwise, we check if the interval corresponds to the queries' one (line 14), if not, a new matching block is initialized (line 15-16). In any case, current pitch $p$ is added to the matching pitches (line 17); as a new occurrence (preceding reset), as a pitch in the same block (same height), or as a following pitch (matching interval).

# 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 of the result list the ones that are closest to $P$. For the sake of illustration, we will now assume that the search pattern is our previous $F$ (Fig. 3, 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 a sequence of 6 intervals $< 6, -3; -3; 1; 2; -2 >$. Intuitively $P_1$ and $P_2$ (Fig. 5) should be ranked first, and $P_5$ (Fig. 6) should be ranked last.

It is important to note that this ranking does not compare arbitrary fragments, but 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 $\pi$ and* VNORM*, each interval $I_j, j \in [1, n]$ is represented in $F$ by a sequence $< p_1^i, e_2^i, \cdots, e_{k-1}^i, p_k^i >$ such that:*

- *$p_1^i$ and $p_k^i$ are two pitches, and $interval(p_1^i, p_k^i) = I_i$*
- *each $e_l^i, l \in [2, k-1]$ is either a rest, or a pitch such that $e_l^i = p_1^i$*

*We call $< p_1^i, e_2^i, \cdots, e_{k-1}^i >$ 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. 8 for $P_1$, $P_3$ and $P_5$.

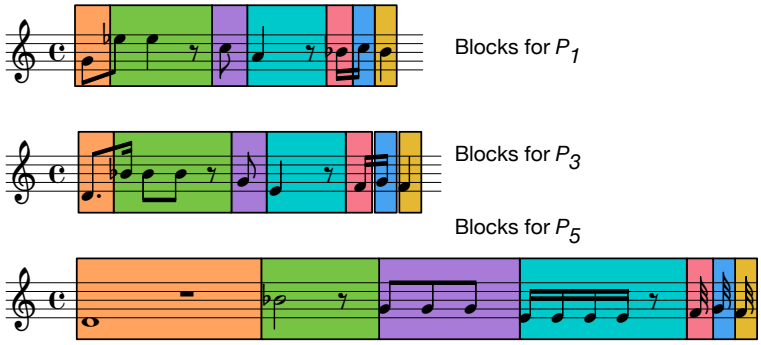



*Figure 8. Blocks for $P_1$, $P_3$, and $P_5$*

The first level of the ranking function evaluates the similarity of two fragments $F_1$ and $F_2$ by comparing the pairwise durations of their blocks. The rationale is that if these durations are similar, the only difference lies in either repeated notes or rests inside each block. Fig. 8 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 function that computes this similarity is simple and efficient. We first normalize the fragment duration, and sum up the difference of durations between corresponding pairs of blocks.

**Algorithm 3** Ranking procedure

1: **procedure** RANKING($F_1, F_2$)
2: **Input:** $F_1$, $F_2$, such that $\pi(\text{VNORM}(F_1)) = \pi(\text{VNORM}(F_2)) =< I_1, \cdots, I_n >$
3: **Output:** a similarity $s \in [0, 1]$
4: $s \leftarrow 0$
5: $d_1 \leftarrow duration(F_1); d_2 \leftarrow duration(F_2)$
6: **for** $i := 0$ **to** $n$ **do** ▷ Loop on the blocks
7: $s \leftarrow s + |dur(B_i^1)/d_1 - dur(B_i^2)/d_2|$
8: **if** $s = 0$ **then** $s \leftarrow TieBreaking(F_1, F_2)$
9: **return** $s/2$

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 a `ngram` 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.

**Example 3.** *If we can encode positive integers with* `a`*,* `b`*,* `c`*, etc., and the minus sign by* `m`*, we would obtain, for the list of* $n$*-grams* $\{< 6, -3; -3 >, < -3; -3, 1 >, < -3, 1, 2 >, < 1; 2; -2 >\}$*, the character string*

*`fmcmc mcmca mcab abmb`*

Those strings are put in the `ngram` field and indexed by ELASTICSEARCH as any regular text. Obviously, more general and efficient encodings exist.

## 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. Taking the pattern $F$ of our running example, we apply the very same transformation and compose the following query:

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

The search engine then does the rest of the job for us. It finds all the indexed documents such that the `ngram` 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 (and actually in any search engine that relies on the Lucene library)Zhou et al. (2008); Travers (2012). To this end, we must provide a Java function that implements the ranking method exposed in Section 5. ELASTICSEARCH calls this function at query time over the result set and produces a score for each according to the similarity function. The result is sorted on this score, and made accessible to the client application via an iterator-like mechanism ElasticSearch (2018) 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. This is the purpose of the `sequence` field.

Basically, we encode in JSON a sequence of the music items that constitute a voice. An excerpt is shown below where each item features the octave, pitch and duration.

```
[ {"o":4, "id":"m60", "s":"A", "a":0, "d":8.0},
  {"o":4, "id":"m83", "s":"A", "a":0, "d":8.0} ]
```

Note that we also store the id of each item, which is actually the id of the corresponding element in the XML encoding of the score[1]. The ranking function integrates in the query result the list of matching occurrences represented as sequences of such item ids. The client application can then highlight each occurence.

**Note**: our system is available on-line, but we cannot disclose its address due to anonymization requirements. In case of acceptance, readers will be able to test on-line the searching, ranking and highlighting features.

### 6.4 Query expansion

Our method relies on a strict matching of a sequence of intervals. We must be very careful if we wish to add some flexibility here, because we would likely return the whole database for each query if we accept unbounded melodic transformations. We can, however, 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 match "car" and "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'. We can therefore give every similar patterns that could match, in order to give more flexibility during the matching process.

The following ngrams are then considered to be similar to the given query pattern. Any similar interval produces a new ngram combination.

```
cbh, dbh, dbi, cbi
```

In order to integrate the list of synonyms to ELASTICSEARCH, the list is given as an analyzer *"melodic_transformation"* to the index and matching inverted lists are merged at query time. To take into account this new analyzer, the query is modified to:

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

To find the matching occurrences, we need to modify the algorithm in order to integrate the synonyms. For this, the computation of intervals for major third and sixth can be authorized, then: 1,5 $\sim$ 2 and 3 $\sim$ 3,5. So, line 14 of Algorithm 2 is modified to integrate those similarities.

This synonym feature can be enhance by taking into account a similarity measure between those synonyms. It is possible to give an oriented graph weighted with similarity values between every ngram synonyms. Then, the scoring function will computing a similarity score based on those weighted similarities. We wish to show that ELASTICSEARCH provides a framework that helps to produce new similarity measures on our data model.

### 6.5 Implementation

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

The first step consists in a Python scripts that normalizes voices from scores (Algorithm 1), extracts corresponding ngrams, and produce a JSON document for each score that is sent to the ELASTICSEARCH REST API. This document also contains corpus and opus ids, syllables from lyrics voices which can be queried to get more relevant and complex queries.

[1] ANONYMIZED currently uses MEI, because the current version of MusicXML does not supply element ids.

**Nicolas**: *Ce n'est peut être pas utile que je rajoute un document JSON stocké (ngram + voix) ?*

To achieve the second step, the `ScoreSim` scoring module has been implemented in Java in order to import it in ELASTICSEARCH. For this, a *SearchScript* needs to be inherited in order to produce a plugin for ELASTICSEARCH. This plugin takes queries' parameters and instanciates a scoring function which will process every matching scores. The scoring function `ScoreSim` implements Algorithms 2 (finding block occurrences) and 3 (producing scores for ranking).

In order to take into account synonyms in ELASTICSEARCH, the list of ngram synonyms needs to be given to the REST API[2] offline. We have generated the list of all combinations of third and sixth transformation for each possible ngram available in the repository. This list is imported in ELASTICSEARCH and then processed on-the-fly for each query.

The following query integrates every features that are proposed in our approach: ngram 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 `ScoreSim` in order to produce the score value from Algorithm 3.

```
{"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}]
}}}}]}}}
```

To illustrate the querying process Figure 9 shows 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 ngram 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.

### 6.6 Performance

In order to study the impact of our approach on the computation time, we apply different queries on our corpora. This corpora is composed of several corpus which represents 4,950 scores in a whole. We will vary the corpus size by cumulating them.

To study the effect of the matching process, we have chosen four different queries to apply with various patterns, from unfrequent to more frequent ones. The four queries were chosen based on the popularity of the stored patterns (ngrams). Table 1 gives the query patterns, 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 amount of matches (2.5 times more). At the opposite, query q3 has no synonyms and do not enlarge its result-set.

Figure 10 shows the evolution of the number of matching scores wrt. the corpus size. It gives both matching scores for normal queries (plain lines) and expanded queries (dashed lines). Each

[2] https://www.elastic.co/guide/en/elasticsearch/guide/current/using-synonyms.html

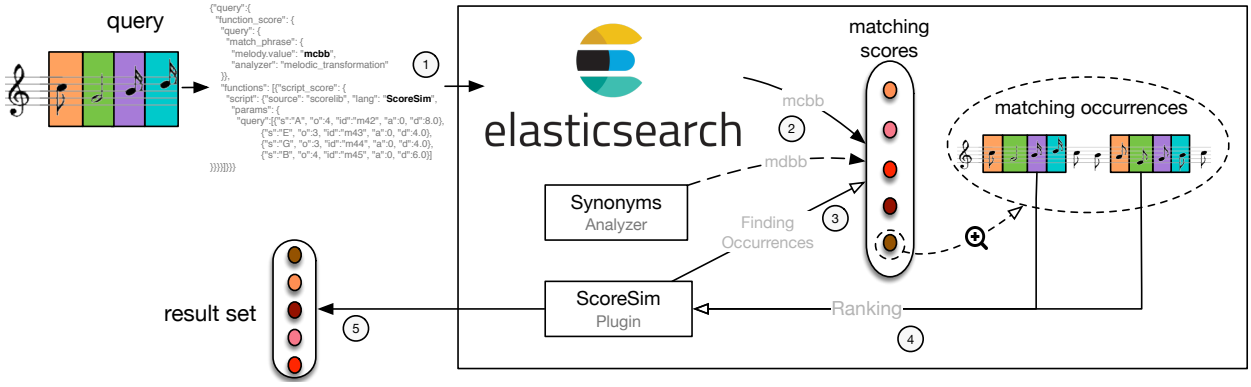



*Figure 9. `ScoreSim` integration in* ELASTICSEARCH

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

*Table 1. Query patterns*

query follows a specific ratio of matchings and is globally homogeneous all over the corpora. According to the query expansion, we can see that q1 and q4 provides few more matchings, while q2 witnesses a really different behavior where the number of matchings grows drastically due to the number of synonyms. It will help to see how the query expansion impacts performances.

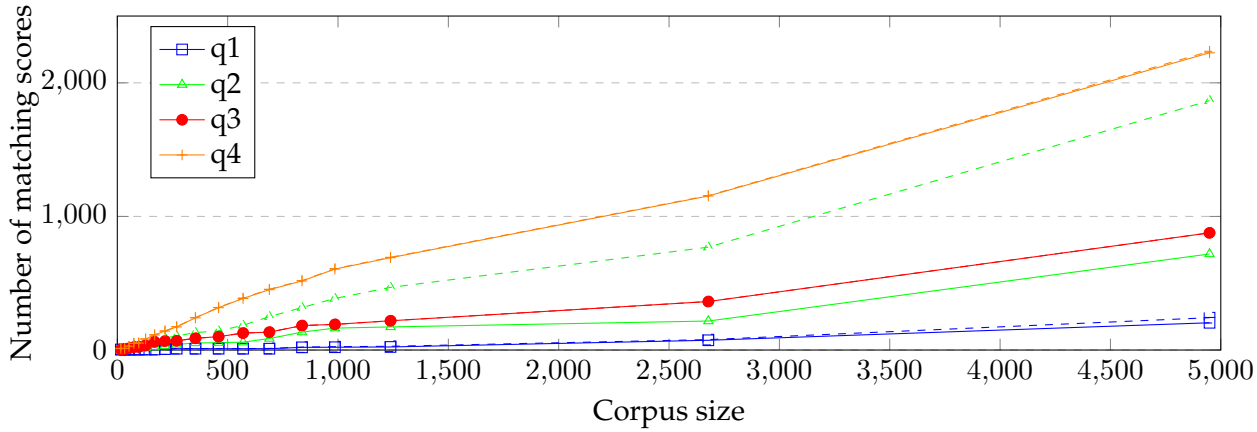



*Figure 10. Number of matching scores with and without query expansion (synonyms)*

The execution time is plotted in Figure 11 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 (almost 45% of the corpora). It is executed in 277 ms. The small number of synonyms has few impacts on the gobal processing time. At the opposite, q1 is extremely efficient due to its 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).

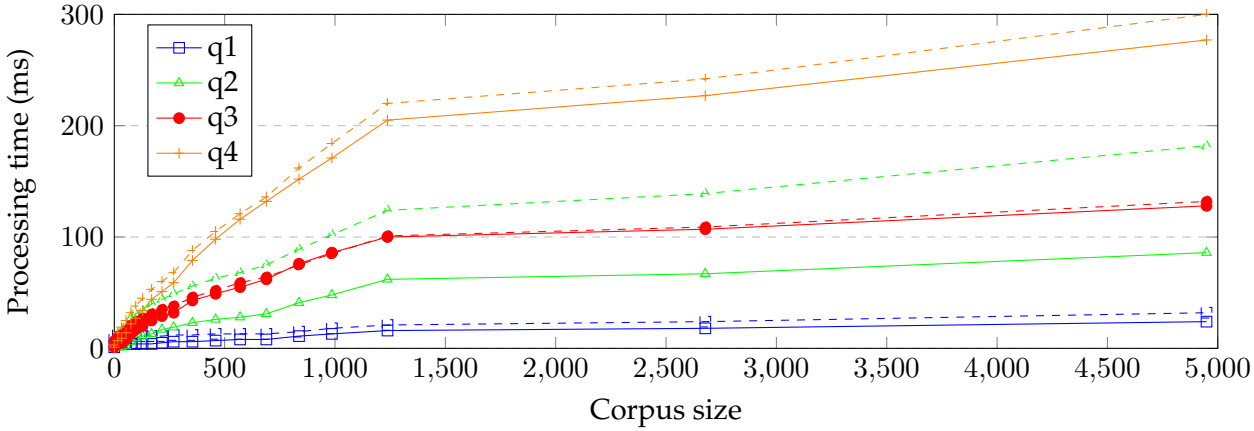



*Figure 11. Execution time with respect to corpus size with and without query expansion*

To study the time spent per score, Figure 12 gives in log scale the average time to process each score with respect to the corpus size. We can see that an initial cost is visible at the beginning of the curves, this constant penalty is associated to the initialization of the `ScoreSim` user-defined similarity function and levels up the average computation time. For small corpus, computing a score depends mostly on the number of occurrences found in voices (q1) which can vary from one corpus to another. Fortunately this effect is smoothed for bigger corpus even for frequent patterns or expanded queries, the ranking function takes less than 0.12ms to process each score.

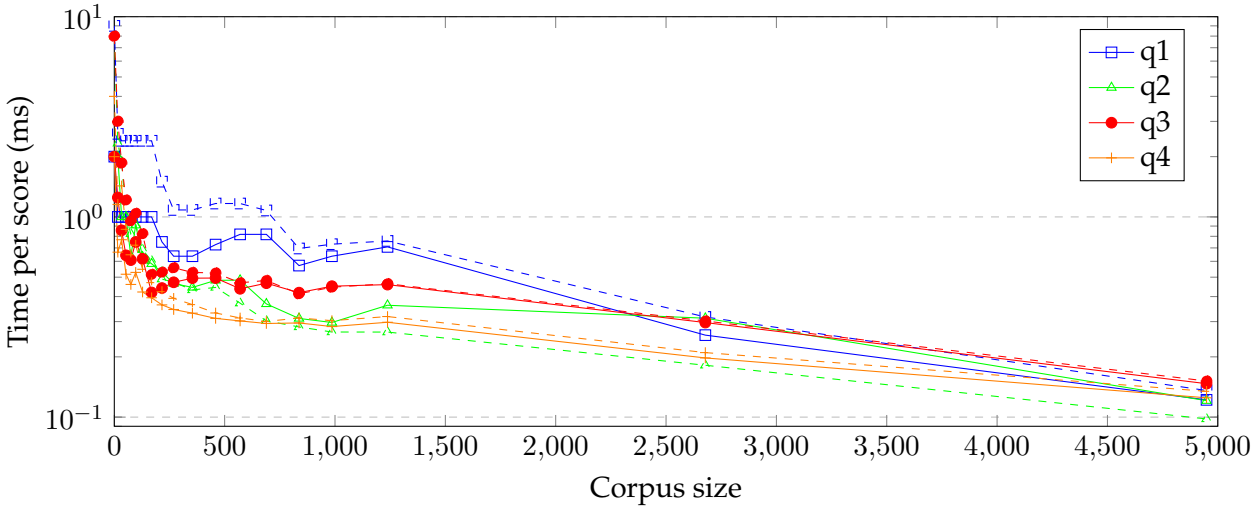



*Figure 12. Execution time per score with respect to corpus size with and without query expansion*

## 7 Related work and discussion

Our approach combines similarity searches based on textual music encoding, scalable search, and rhythm-based ranking. None of these techniques, considered separately, is completely new. But their association in a consistent setting, and the trivial implementation in a standard system make, in our opinion, our solution quite attractive.

### 7.1 Related Work

Music similarity has been an active MIR research topic over the last decades Casey et al. (2008). The general goal is to evaluate the likeness of two musical sequences, in a way that matches as much as possible the human listener perception. A major problem raised by this definition is that similarity judgements are highly dependent on both the content being compared and on the user taste, culture, and experience Jones et al. (2007); Ens et al. (2017). This encourages a proliferation of methods, often originated from different fields, and makes difficult their comparison. The following is a brief overview that positions our approach. The interested reader is referred to the recent survey Velardo et al. (2016) which summarizes the recent trends observed in the SMS track of the MIREX competition.

The selection of musical parameters is probably one of the most important choices that characterize a similarity method. We rely on pitches and durations, which 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. This eliminates more complex representations where the price to pay is a less expressive matching semantic. Some important parameters, e.g., metric accent, structure or harmonic are ignored because they most often lead to tree-based encodings that are much less easily indexable. Geometric approaches, such as Typke et al. (2003), are also less suitable in this indexing perspective. Multidimensional structures are complex, and their performances are known to fall down as the dimension increases Samet (2006). Moreover, they have not yet found their in off-the-shell search engines.

Textual encoding of symbolic music representation is an old and attractive idea because standard text algorithms can be used. The HumDrum toolkit Knopke (2008) relies on a specialized text format and adapts Unix file inspection tools for music analysis purposes. Exact and approximate string matching algorithms for melody matching have been used in ThemeFinder Kornstaedt (1998) or Musipedia Prechelt and Typke (2001). Many algorithms for efficient computation of similarity matching through exhaustive searches have been proposed Cantone et al. (2004, 2005); Clifford and Iliopoulos (2004); Cambouropoulos et al. (2005). Specialized rhythm similarity functions are proposed and compared in Toussaint (2004).

Text-based approaches are simple solutions, with two important limitations. First, these methods are designed for text and are not well adapted to the specificities of music representation. Combining the two main dimensions (pitches and rhythm) is a single character string for instance is not easy, and small music variants may result in important syntactic differences. Since, in addition, we usually search for a match between a query pattern and any subsequence of a voice encoding, it is unclear which substrings have to be chosen before applying the approximate matching operation.

Second, these methods do not scale since the whole database has to be inspected for each query. Text matching algorithms are not always easily supported by an index. In the specific context of the edit distance, several indexing methods have been suggested, an overview of which can be found in Navarro et al. (2001). The Dynamic Time Warping distance is another popular method, for which sophisticated indexing structures have been proposed Keogh and Ratanamahatana (2005). 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 Downie (1999); Neve and Orio (2004); Doraisamy and Rüger (2004); Chang and Jiau (2006). Each $n$-gram plays the role of a "token" and standard search methods apply. Keywords search will retrieve a sequence even though a subset of the $n$-gram match those of the patterns. This is somewhat similar to a similarity search. Phrase search, on the other hand, corresponds to an exact search of the indexed sequence. The

flexibility lies in the pre-processing transformation applied to both the stored sequence and the query pattern.

Ranking is an essential part of an information retrieval system. Any query with a few keywords retrieves from the Web millions of documents. The quality of the result is mostly estimated by the ability of the search engine to top-rank the most relevant ones. We believe that our proposal, which combines a pre-processing normalization step, a search phase based on the melodic profile and a ranking based on the rhythmic profiles, 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 to its users. We are currently working on an open-source release of our packages on GitHub.

The proposal presented in this paper is by no way a final achievement. Several extensions can be envisaged. Let us mention the ones that seem most promising at the time of writing.

**Score analyzers.** Currently, the main transformation is the normalization of voices described by Algorithm 1. However, several others are possible with regards to, e.g., the management of grace notes, the simplification of melodic profiles, or how we treat repeated notes, to name a few. Search engines propose numerous methods of text processing that can be combined in order to form a dedicated *analyzer*. We can do the same thing for music indexing, letting the user define ad hoc music analyzers depending on her needs.

# References


Cambouropoulos, E., M. Crochemore, C. S. Iliopoulos, M. Mohamed, and M.-F. Sagot. 2005. "A Pattern Extraction Algorithm for Abstract Melodic Representations that Allow Partial Overlapping of Intervallic Categories." In *Proc. Intl Conf. on Music Information Retrieval (ISMIR)*. pp. 167–174.

Cantone, D., S. Cristofaro, and S. Faro. 2004. "Efficient Algorithms for the delta-Approximate String Matching Problem in Musical Sequences." In *Proc. of the Prague Stringology Conf. (Stringology)*. pp. 33–47.

Cantone, D., S. Cristofaro, and S. Faro. 2005. "Solving the $(\delta, \alpha)$-Approximate Matching Problem Under Transposition Invariance in Musical Sequences." In *Proc. Intl Conf. on Music Information Retrieval (ISMIR)*. pp. 460–463.

Casey, M., R. Veltkamp, M. Goto, M. Leman, C. Rhodes, and M. Slaney. 2008. "Content-based music information retrieval: current directions and future challenges." *PROCEEDINGS OF THE IEEE* 96(4):668–696.

Chang, C.-W., and H. C. Jiau. 2006. "An Efficient Numeric Indexing Technique for Music Retrieval System." In *Proc. IEEE Intl Conf. on Multimedia and Expo (ICME)*. pp. 1981–1984.

Clifford, R., and C. S. Iliopoulos. 2004. "Approximate string matching for music analysis." *Software Computing* 8(9):597–603.

Doraisamy, S., and S. M. Rüger. 2004. "A Polyphonic Music Retrieval System Using N-Grams." In *Proc. Intl Conf. on Music Information Retrieval (ISMIR)*. pp. 204–209.

Downie, J. 1999. "Evaluating a simple approach to music information retrieval: Conceiving melodic n-grams as text." Ph.D. thesis, Univ. Western Ontario.

ElasticSearch. 2018. "Advanced scripts using script engines." https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-scripting-engine.html.

Ens, J., B. E. Riecke, and P. Pasquier. 2017. "The Significance of the Low Complexity Dimension in Music Similarity Judgements." In *Proc. Intl. Conf. on Music Information Retrieval (ISMIR)*. pp. 31–38.

Good, M. 2001. *The Virtual Score: Representation, Retrieval, Restoration*, chapter "MusicXML for Notation and Analysis". W. B. Hewlett and E. Selfridge-Field, MIT Press, pp. 113–124.

Jones, M. C., J. S. Downie, and A. F. Ehmann. 2007. "Human Similarity Judgments: Implications for the Design of Formal Evaluations." In *Proc. Intl. Conf. on Music Information Retrieval (ISMIR)*. pp. 539–542.

Keogh, E. J., and C. A. Ratanamahatana. 2005. "Exact Indexing of Dynamic Time Warping." *Knowl. Inf. Syst.* 7(3):358–386.

Knopke, I. 2008. "The Perlhumdrum and Perllilypond Toolkits for Symbolic Music Information Retrieval." In *Proc. Intl. Conf. on Music Information Retrieval (ISMIR)*. pp. 147–152.

Kornstaedt, A. 1998. "Themefinder: A Web-based Melodic Search Tool." *Computing in Musicology* 11:231–236.

Manning, C. D., P. Raghavan, and H. Schutze. 2008. *Introduction to Information Retrieval*. Cambridge University Press. Online version at *http://informationretrieval.org/*.

MEI. 2015. "Music Encoding Initiative." http://music-encoding.org. Accessed March 2017.

MNX. 2018. "Music Notation Community Group." https://www.w3.org/community/music-notation/. Accessed March 2018.

Navarro, G., R. Baeza-yates, E. Sutinen, and J. Tarhio. 2001. "Indexing Methods for Approximate String Matching." *IEEE Data Engineering Bulletin* 24(4):19–27.

Neve, G., and N. Orio. 2004. "Indexing and Retrieval of Music Documents through Pattern Analysis and Data Fusion Techniques." In *Proc. Intl Conf. on Music Information Retrieval (ISMIR)*. pp. 216–223.

Prechelt, L., and R. Typke. 2001. "An Interface for Melody Input." *ACM Trans. Comput.-Hum. Interact.* 8(2):133–149.

prince, J. 2014. "Contributions of pitch contour, tonality, rhythm, and meter to melodic similarity." *journal of experimental psychology.* 40(6):2319–2337.

Rolland, P. 2002. "The Music Encoding Initiative (MEI)." In *Proc. Intl. Conf. on Musical Applications Using XML*. pp. 55–59.

Samet, H. 2006. *Multidimensional and Metric Data Structures*. Morgan Kaufmann.

Toussaint, G. 2004. "A comparison of rhythmic similarity measures." In *Proc. Intl. Conf. on Music Information Retrieval (ISMIR)*. pp. 242–245.

Travers, N. 2012. *Web Data Management*, chapter Putting into Practice: Full-Text Indexing with LUCENE. Cambridge University Press, pp. 355–363.

Typke, R., P. Giannopoulos, R. C. Veltkamp, F. Wiering, and R. van Oostrum. 2003. "Using transportation distances for measuring melodic similarity." In *Proc. Intl. Conf. on Music Information Retrieval (ISMIR)*.

Velardo, V., M. Vallati, and S. Jan. 2016. "Symbolic Melodic Similarity: State of the Art and Future Challenges." *Computer Music Journal* 40:70–83.

Zhou, C., B. Feng, and Z. Li. 2008. "Research and Implementation of the Small-Scale Search Engine Based on Lucene." In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*, CSSE '08. IEEE Computer Society, pp. 377–380.