QUERYING XML SCORE DATABASES: XQUERY IS NOT ENOUGH!

Raphaël Fournier-S'niehotta
CNAM
fournier@cnam.fr

Philippe Rigaux
CNAM
philippe.rigaux@cnam.fr

Nicolas Travers
CNAM
nicolas.travers@cnam.fr

ABSTRACT

The paper addresses issues related to the design of query languages for searching and restructuring collections of XML-encoded music scores. We advocate against a direct approach based on XQuery, and propose a more powerful strategy that first extracts a structured representation of music notation from score encodings, and then manipulates this representation in closed form with dedicated operators. The paper exposes the content model, the resulting language, and describes our implementation on top of a large Digital Score Library (DSL).

1. INTRODUCTION

It is now common to serialize scores as XML documents, using encodings such as MusicXML [11, 16] or MEI [15, 18]. Ongoing work held by the recently launched W3C Music Notation Community Group [20] confirms that we can expect in a near future the emergence of large collections of digital scores.

1.1 Issues with Querying XML score databases

A natural question in a collection management perspective is the definition of a query and manipulation language to access, search, and possibly analyze these collections. While XQuery appears as a language of choice, we consider that it does not constitute, as such, a suitable solution. There are several reasons that prevent XQuery from being able to address the complex requirements of music notation manipulation, at least beyond the simplest operations.

1. **Issue 1: Heterogeneity.** Score encodings closely mix information related to the *content* (e.g., the sequence of notes of a voice) and to a specific *rendering* of this content (e.g., voice allocation to a staff, positioning of notes/lines/pages, and other options pertaining to scores visualization). While it is not always obvious to clearly distinguish content from rendering instructions, mixing both concerns leads to an intricate encoding from which selecting the relevant information becomes extremely difficult. Extracting for instance melodic information from either MusicXML or MEI turns out to be difficult; and more sophisticated extractions (e.g., alignment of melodic information for voices) are almost impossible.

2. **Issue 2: Operations and closure.** One of the main requirements of a query language is a set of well-defined operations that operate in closed form: given as input objects that comply to some structural constraints (a data model), the language should guarantee that any output obtained by combining the operators is model-compliant as well. This requirement allows an arbitrary composition of operations, and ensures the safety of query results. In our context, it concretely means that we need operators that manipulate music notation, and that their output should consist of valid music notation as well. There is however no means to achieve that with XQuery, nor even to simply know whether a query supplies a compliant output or not.

3. **Issue 3: Formats.** Finally, a last, although less essential, obstacle is the number of possible encodings available nowadays, from legacy formats such as HumDrum to recent XML proposal mentioned above. Abstracting away from the specifics of these formats in favor of the core information set that they commonly aim at representing would allow to get rid of their idiosyncrasies.

To summarize, we consider that XML representation of scores are so far mostly intended as a serialization of documents that encapsulate all kinds of information, from metadata (creation date, encoding agent) to music information (symbolic representation of sounds and sounds synchronization) via rendering instructions. They are by no means designed to supply a structured representation of a core aspect (music "content"), subject to investigation and manipulation via a dedicated, model-aware query language.

1.2 Our approach

We make the case for an approach that leverages music content information as virtual XML objects. Coupled with a specialization of XQuery to this specific representation, we obtain a system apt at providing search, restructuration, extraction, analytic services on top of large collections of XML-encoded scores. The system architecture is summarized in Figure 1, and addresses the above issues.

© Raphaël Fournier-S'niehotta, Philippe Rigaux, Nicolas Travers. Licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). **Attribution:** Raphaël Fournier-S'niehotta, Philippe Rigaux, Nicolas Travers. "QUERYING XML SCORE DATABASES: XQUERY IS NOT ENOUGH!", 17th International Society for Music Information Retrieval Conference, 2016.

Issue 1: Bringing homogeneity. The bottom layer is a Digital Score Library managing collections of scores serialized in MusicXML, MEI, or any other legacy format (e.g., Humdrum). This encoding is *mapped* toward a model layer where the content structured in XML corresponds to the model structures. This defines, in intention, collections of music notation objects that we will call *vScore* in the following. A vScore abstracts the part of the encoding (here the content) we wish to focus on, and gets rid of information considered as useless, at least in this context.

Issue 2: Defining a domain-specific language. In order to manipulate these vScores, we equip XQuery with two classes of operations dedicated to music content: *structural operators* and *functional operators*. The former implement the idea that structured scores management corresponds, at the core level, to a limited set of fundamental operations, grouped in a score algebra, that can be defined and implemented only once. The latter acknowledges that the richness of music notation manipulations calls for a combination of these operations with user-defined functions at early steps of the query evaluation process. Modeling the invariant operators and combining them with user-defined operations constitutes the operational part of the model. This yields a query language whose expressions unambiguously define the set of transformations that produce new *vScores* from the base collections.

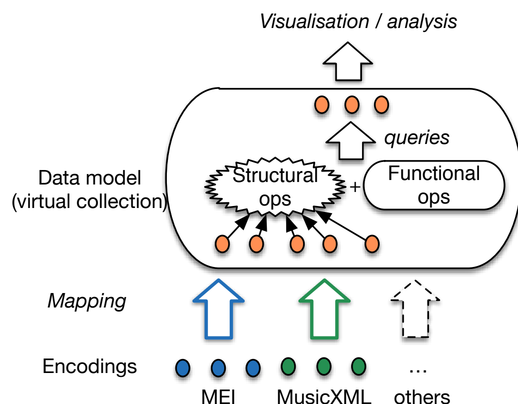

Figure 1. Envisioned system

Issue 3: Serialization independence. One mapper has to be defined for each possible encoding, as shown by the figure which assumes that MusicXML and MEI documents cohabit in a single DSL. Adding a new source represented with a new encoding is just a matter of adding a new mapper. Each document in the DSL is then mapped to a (virtual) XML document, instance of the model.

1.3 Contributions

In the present paper, we describe the implementation of the above ideas in NEUMA [17]. The focus is on the model layer, defined as a virtual XML schema, on the mapping from raw documents to vScores, and on the integration of XQuery with structural operators and external functions. The score algebra, not presented here, is implemented in our system as XQuery functions, whose meaning should be clear from the context.

Section 2 gives the virtual XML schema for music content notation, and Section 3 shows how to create an XML database referring to vScores. Section 4 presents the query language and Section 5 discusses salient implementation choices. Section 6 covers related work and Section 7 concludes the paper.

2. MUSIC NOTATION: THE SCHEMA

We now describe the virtual data model with XML Schema [22]. The model aims at representing polyphonic scores in Common Music Notation (CMN). A score is composed of voices, and a voice is a sequence of events.

2.1 Event type

An event is a value (complex or simple) observed during a time interval. Events are polymorphic: the value may be a note representation, a chord representation, a syllable or any other value (e.g., an integer representing an interval).

The abstract definition of an event is a complex type with a duration attribute.

```
<xs:complexType abstract="true" name="eventType">
  <xs:attribute type="xs:integer" name="duration"
       use="required"/>
</xs:complexType>
```

From this abstract type, we can derive concrete event types with specific element names. The most important are events denoting sounds, which covers simple $n \geq 0$ simultaneous sounds, either rests ($n = 0$), notes ($n = 1$) or chords ($n > 1$). The `soundType` is derived from the `eventType` as follows:

```
<xs:complexType name="soundType">
  <xs:complexContent>
    <xs:extension base="eventType">
      <xs:choice minOccurs="1" maxOccurs="1">
        <xs:element name="note" type="noteType"/>
        <xs:element name="rest" type="restType"/>
        <xs:element name="chord" type="chordType"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Due to space restrictions, we do not detail `noteType` (containing 'pitch' and 'octave' attributes), `restType` (empty element) and `chordType` (list of `noteType`). As another example of a concrete event type, lyrics can be represented with syllabic events (and rests), with type:

```
<xs:complexType name="syllableType">
  <xs:complexContent>
    <xs:extension base="eventType">
      <xs:choice minOccurs="1" maxOccurs="1">
        <xs:element name="syll" type="xs:string"/>
        <xs:element name="rest" type="restType"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Events are not restricted to musical domains. Events in the `xs:integer` domain, for instance, can be used to represent intervals, obtained by a 2-voices scores analysis.

```
<monody>
  <rest duration="24"/>
  <note duration="16" p="D" o="5"/>
  <rest duration="4"/>
  <note duration="2" p="E" o="5"/>
  <note duration="2" p="F" o="5"/>...
</monody>
```

```
<lyrics>
  <rest duration="24"/>
  <syll duration="16"/>Ah,</syll>
  <rest duration="4"/>
  <syll duration="2"/>que</syll>
  <syll duration="2"/>je</syll>...
</lyrics>
```

```
<bass>
  <note duration="8" p="D" o="4"/>
  <note duration="4" p="C" o="4"/>
  <chord duration="4">
    <note p="D" o="4" a="-1"/>
    <note p="B" o="3" a="-1"/></chord>
  <note duration="4" p="A" o="3"/>
  <note duration="4" p="G" o="3"/>...
</bass>
```

Figure 2. Voices representation

2.2 Voice type

A voice is a sequence of events. Its abstract definition is given by the following schema:

```
<xs:complexType name="voiceType" abstract="true">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element type="eventType"/>
  </xs:sequence>
</xs:complexType>
```

This type actually represents a function from the time domain to the domain of events. There is an implicit non-overlapping constraint: an event begins when its predecessor ends. We can instantiate concrete voice types by simply replacing the abstract `eventType` by one of its derived types (e.g., `soundType`, `syllableType`, `intType`), like in the following example:

```
<xs:complexType name="lyricsType">
 <xs:extension base="voiceType">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
   <xs:element type="syllableType"/>
  </xs:sequence>
 </xs:extension>
</xs:complexType>
```

2.3 Score type

Finally, our virtual notation schema contains a score type which describes a recursive structure defined as follows:

- if v a voice, then v is a score.
- if $s_1, \cdots, s_n$ are scores, the sequence $< s_1, \cdots, s_n >$ is a score.

The generic definition of the XML schema for this structure is given below. Note that element names for scores and voices will be specified for each specific corpus.

```
<xs:complexType name="scoreType" abstract="true">
  <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:choice>
      <xs:element type="scoreType"/>
      <xs:element type="voiceType"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```

2.4 Example

Let's illustrate by an example our types and structure. Fig. 3 is partially represented by the vScore in Fig. 2.

Figure 3. A score example

Our model decomposes the score in three voices. The first one represents the monody of the vocal part. It consists of a sequence of `soundType` events. The second voice represents lyrics with `syllableType` events. And, finally, the last voice is the bass. Its representation is a sequence of `soundType` events (here, notes and chords).

The vScore itself illustrates the recursive structure that encapsulates the voices in a tree of `score` elements. The upper level combines the `bass` voice and an embedded `score` which combines the `monody` and `lyrics` voices. The structure is as follows.

```
<air>
  <vocal>
    <monody>(...)</monody>
    <lyrics>(...)</lyrics>
  </vocal>
  <bass>(...)</bass>
</air>
```

It should be clear that this representation abstracts a part of the content that can be found in all the encodings we are aware of. The choice of the information subset which is selected is here minimal, for the sake of conciseness. We can obviously extend the representation with additional details as long as it does not affect the structure. A voice can be "decorated" by an instrument name, an event by the current metric or the measure number, a score by its composer, all represented as additional elements. In general, the issue relates to what is considered as "content" subject to search and analysis operations, and what is the suitable representation for this content. We will stick in the following to the simple model given above which is sufficient to our needs.

Recall that the schema intends to define a *virtual* score representation which is derived at search time (according to rules explained in the next sections) from the actual serialization. We briefly explain the mapping from MusicXML or MEI documents to vScores.

2.5 Mapping from MusicXML or MEI

In MusicXML, scores are organized as a tree of `score`, `part-group`, and `part` elements. Voices are numbered with respect to the part they belong to, and represented as nested elements of notes, rest and chords. Our mapping unifies the score, groups and parts in a recursive nesting of `score` elements and (virtually) splits the music and lyrics as two associated voices.

In MEI, the score structure is based on score, staves and groups of staves. Voices are represented as `layer` objects deeply nested in a hierarchy of `measure` and `staff` containers. Our mapping extracts the voice events from the complex imbrication of MEI elements or organizes them according to our recursive `score` structure.

3. MUSIC NOTATION: THE DATABASE

XML databases are collections of documents. Although the definition of a schema for a collection is not mandatory, it is a safe practice to ensure that all the documents it contains share a similar structure. In our context, a collection of digital scores is a regular XML collection where one or several elements are of `scoreType` type. Here is a possible example:

```
<xs:complexType name="opusType">
  <xs:sequence>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="composer" type="xs:string"/>
    <xs:element name="published" type="xs:string"/>
    <xs:element type="scoreType"/>
  </xs:sequence>
  <xs:attribute type="xs:ID" name="id"/>
</xs:complexType>
```

Note that this schema is strict regarding meta-data (title, composer) but very flexible for the music notation because we are using here the generic `scoreType`. It follows that, from one document to the other in standard notations (musicXML, MEI, HumDrum), the score structure, the number of voices and their identifiers may vary. Such a flexibility is definitely not convenient when it comes to querying a collection, since we cannot safely refer to the components of a score. It seems more appropriate to base the organization of score collections on an homogeneous score structure. A *Quartet* collection for instance would only accept scores complying to the following structure:

```
Quartet(id: int, title: string,
        composer: string, published: date,
        music: Score [v1, v2, alto, cello])
```

The XML Schema formalism accepts the definition of *restriction* of a base type. In our case, the restriction consists in specializing the `scoreType` definition to list the number and names of voices, like in `quartetType`:

```
<xs:complexType name="quartetType">
 <xs:complexContent>
  <xs:restriction base="scoreType">
   <xs:sequence>
    <xs:element name="v1" type="soundVoiceType"/>
    <xs:element name="v2" type="soundVoiceType"/>
    <xs:element name="alto" type="soundVoiceType"/>
    <xs:element name="cello" type="soundVoiceType"/>
   </xs:sequence>
  </xs:restriction>
  <xs:attribute type="xs:ID" name="id"/>
 </xs:complexContent>
 </xs:complexType>
```

Using `QuartetType` in place of `ScoreType` in the collection schema ensures that all the vScores in the collection match the definition. Voices' names are specified and can then be used to access to the (virtual) music notation to start applying operations and transformations. This can be done with XQuery, as explained in the next section.

4. XQUERY + SCORE ALGEBRA = QUERIES

With a well-defined collection and a clear XML model to represent the notation of music content, we can now express queries over this collection with XQuery. However, executing such queries gives rise to the following issues:

Issue 1: We *cannot* directly evaluate an XQuery expression, since they are interpreted over instances which are partly virtual (scores, voices and events) and partly materialized (all the rest: title, composer, etc.).
Issue 2: Pure XQuery expression would remain limited to exploit the richness of music notation;

The first issue is solved by executing, at run-time, the mapping that transforms the serialized score (say, in MusicXML) to a vScore, instance of our model. This is further explained in the next section, devoted to implementation choices. To solve the second issue, we implemented a set of XQuery functions forming a score algebra. We introduce it and illustrate the resulting querying mechanism with examples.

4.1 Designing a score algebra

We designed a score algebra in a database perspective, as a set of operators that operate in closed form: each takes one or two vScores (instances of `scoreType`) as input and produces a vScore as output. This brings composition, expressiveness, and safety of query results, since they are guaranteed to consist of vScore instances that can, if needed, be serialized back in some standard encoding (see the discussion in Section 1 and the system architecture, Fig. 1). The algebra is formalized in [8], and implemented as a set of query functions whose meaning should be clear from the examples given next.

4.2 Queries

The examples rely on the *Quartet* corpus (refer to the Section 3 for its schema). Our first example creates a list of *Haydn*'s quartets, reduced to the titles and violin's parts.

```
for $s in collection("Quartet")
where $s/composer="Haydn"
return $s/title, Score($s/music/v1, $s/music/v2)
```

Recall that `music` is a `QuartetType` element in the *Quartet* schema. This first query shows two basic operators to manipulate scores: projection on voices (obtained with XPath), and creation of new scores from components (voices or scores) with the `Score()` operator.

A third operator illustrated next is MAP. It represents a higher-order function that applies a given function f to each event in a vScore, and returns the score built from f's results. Here is an example: we want the quartets where the v1 part is played by a B-flat clarinet. We need to transpose the `v1` part 2 semi-tones up.

```
for $s in collection("Quartet")
where $s/composer="Haydn"
let $clarinet := Map ($s/music/v1, transpose (2))
let $clrange := ambitus ($clarinet)
return $s/title, $clrange,
       Score($clarinet, $s/music/v2,
             $s/music/alto, $s/music/cello)
```

This second query shows how to define variables that hold new content derived from the vScore via *user defined functions* (UDFs). For the sake of illustration we create two variables, `$clarinet` and `$clrange`, calling respectively `transpose()` and `ambitus()`.

In the first case, the function has to be applied to each event of the violin voice. This is expressed with MAP which yields a new voice with the transposed events. By contrast, `ambitus()` is directly applied to the voice as a whole. It produces a scalar value.

MAP is the primary means by which new vScores can be created by applying all kinds of transformations. MAP is also the operator that opens the query language to the integration of *external* functions: any library can be integrated as a first-class component of the querying system, providing some technical work to "wrap" it conveniently.

The selection operator takes as input a vScore, a Boolean expression e, and filters out the events that do not satisfy e, replacing them by a `null` event. Note that this is different from *selecting* a score based on some property of its voice(s). The next query illustrates both functionalities: a user-defined function "lyricsContains" selects all the psalms (in a *Psalters* collection) such that the vocal part contains a given word ("*Heureux*"), "nullify" the events that do not belong to the measures five to ten, and trim the voice to keep only non-null events.

```
for $s in collection("Psalters")
let $sliced := trim(select ($s/air/vocal/monody,
                           measure(5, 10)))
where lyricsContains ($s/air/vocal/lyrics, "Heureux")
return $s/title, Score($sliced)
```

We can take several vScores as input and produce a document with several vScores as output. The following example takes three chorals, and produces a document with two vScores associating respectively the alto and tenor voices.

```
for $c1 in collection("Chorals")[@id="BWV49"]/music,
    $c2 in collection("Chorals")[@id="BWV56"]/music,
    $c3 in collection("Chorals")[@id="BWV12"]/music
return <title>Excerpts of chorals 49, 56, 12</title>,
         Score($c1/alto, $c2/alto, $c3/alto),
         Score($c1/tenor, $c2/tenor, $c3/tenor)
```

Finally, our last example shows the extended concept of score as a voice synchronization which are not necessarily "music" voices. The following query produces, for each quartet, a vScore containing the violin 1 and cello voices, and a third one measuring the interval between the two.

```
for $s in collection("Quartet")/music
let $intervals := Map(Score($s/v1,$s/cello),interval())
return Score ($s/v1, $s/cello, $intervals)
```

Such a "score" cannot be represented with a traditional rendering. Additional work on visualization tools that would closely put in perspective music fragments along with some computed analytic feature is required.

5. IMPLEMENTATION

Our system integrates an implementation of our score algebra, a mapping that transforms serialized scores to vScores, and off-the-shelf tools (a native XML database, BASEX [1], a music notation library for UDFs, MUSIC21 [2] [4]). This simple implementation yields a query system which is both powerful and extensible (only add new functions wrapped in XQuery/BASEX). We present its salient aspects.

[1] http://basex.org
[2] http://web.mit.edu/music21

5.1 Query processing

The architecture presented in Figure 4 summarizes the components involved in query processing. Data is stored in BASEX in two collections: the semi-virtual collection (e.g., Quartet) of music documents (called *opus*), and the collection of serialized scores, in MusicXML or MEI. Each virtual element `scoreType` in the former is linked to an actual document in the latter.

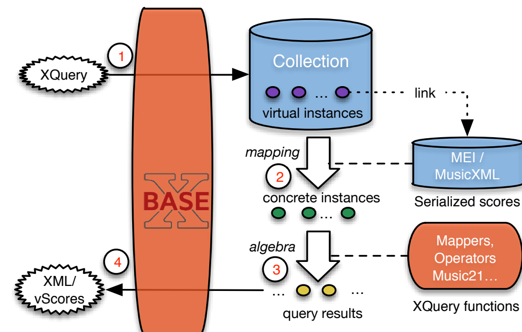

Figure 4. Architecture

The evaluation of a query proceeds as follows. First (step 1), BASEX scans the virtual collection and retrieves the opus matching the `where` clause (at least for fields that do not belong to the virtual part, see the discussion at the end of the section). Then (step 2), for each opus, the embedded virtual element `scoreType` has to be materialized. This is done by applying the mapping that extracts a vScore instance from the serialized score, thanks to the link in each opus.

Once a vScore is instantiated, algebraic expressions, represented as composition of functions in the XQuery syntax, can be evaluated (step 3). We wrapped several Python and Java libraries as XQuery functions, as permitted by the BASEX extensible architecture. In particular, algebraic operators and mappers are implemented in Java, whereas additional, music-content manipulations are mostly wrapped from the Python Music21 toolbox.

The XQuery processor takes in charge the application of functions, and builds a collection of results (that includes instances of `scoreType`), finally sent to the client application (step 4). It is worth noting that the whole mechanism behaves like an ActiveXML [1] document which *activates* the XML content on demand by calling an external service (here, a function).

5.2 Mappers

In order to map scores from the physical representation to the virtual one, references to physical musical parts are matched, according to the collection schema. To achieve this, an ID is associated to each voice element of the materialized score. This ID directly identifies the underlying part of the physical document.

The main mapping challenge is to identify virtual and serialized voices, in particular when they are not standardized according to the collection schema. We need to gen-

erate IDs for each parts on the underlying encoding (MusicXML, MEI, etc.), even for monody and lyrics parts which can be merged on the physical document. Most of them can be done automatically with metadata information given by the collection schema.

5.3 Manipulating vScores with ScoreLib

The ScoreLib Java library is embedded in every BaseX query in order to process our algebraic operations on vScores. The library is responsible for managing the link between the virtual and the physical score, whatever the encoding format. Whenever a function *activates* a vScore by calling a function (whether a structural function of the algebra, or a user-defined function), the link is used to get and materialize the corresponding score.

Once vScore has been materialized, it is kept in a cache in order to avoid repeated and useless applications of the mapping. Temporary vScores produced by applying algebraic operators are kept in the cache as well.

The `Score()` operator creates the final XML document featuring one or several instances of `scoreType`. It combines scores produced by operators and referred to by XQuery variables.

5.4 Indexing music features

User Defined Functions (*UDFs*) are necessary to produce derived information from music notation, and have to be integrated as external components. Getting the highest note of a voice for instance is difficult to express in XQuery, even on the regular structures of our model. In general, getting sophisticated features would require awfully complex expressions. In our current implementation, UDFs are taken from MUSIC21 and wrapped as XQuery functions. This works with quite limited implementation efforts, but can be very inefficient since every score must be materialized before evaluating the UDF. Consider the following example which retrieves all the quartets such that the first violin part gets higher than e6:

```
for $s in collection("Quartet")
where highest($s/music/v1)  > 'e6' return $s
```

A naive, direct evaluation maps the MusicXML (or MEI) document as a vScore, passes it to the XQuery function that delegates the computation to the user library (e.g., MUSIC21 or any other) and gets the result. This has to be done for each score in the collection, even though they do not all match the selection criteria.

A solution is to materialize the results of User Defined Functions as metadata in the virtual document and to index this new information in BaseX. This can directly serve as a search criteria without having to materialize the vScore. The result of the *highest()* function is such a feature. Index creation simply scans the whole physical collections, runs the functions and records it result in a dedicated `index` sub-element of each opus, automatically indexed in BaseX. To evaluate the query above, it uses the access path to directly get the relevant opus.

```
for $s in collection("Quartet")[index/v1/highest  > 'e6']
return $s
```

6. RELATED WORK

Accessing to structured music notation for search, analysis and extraction is a long-term endeavor. Humdrum [13] works on plain text (ASCII) file format, whereas MUSIC21 [4] deals with MIDI channels modeled as musical layers. Both can import widely used formats like MusicXML or MEI. Both are powerful toolkits, but their main focus is on the development of scripts and not database-like access to structured content. As a result, using, say MUSIC21 to express the equivalent of our queries would require to develop ad-hoc scripts possibly rather complex. It becomes all the more complicated when dealing with huge collections of scores. On the other hand, there are many computations that a database language cannot express, which motivated our introduction of UDFs in the language.

Other musical score formalisms rather target generative process and computer-aided composition. This is the case of Euterpea [12] (in `Haskell`), musical programming approaches [3, 6, 7, 14] and operations on *tiled streams* in T-Calculus [14]. They follow the paradigm of abstract data types for music representation, bringing a simplification to the music programming task, but they are not adapted to the conciseness of a declarative query language.

Since modern score formats adopt an XML-based serialization, XQuery [23] has been considered as the language of choice for score manipulation [9]. THoTH [21] also proposes to query MusicXML with patterns analysis. For reasons developed in the introduction, we believe that a pure XQuery approach is too generic to handle the specifics of music representation.

Our work is inspired by XQuery mediation [10, 5, 2, 19], and can be seen as an application of method that combines queries on physical and virtual instances. It borrows ideas from ActiveXML [1], and in particular the definition of some elements as “triggers” that activate external calls.

7. CONCLUSION

We propose in the present paper a complete methodology to view a repository of XML-structured music scores as a structured database, equipped with a domain-specialized query language. Our approach aims at limiting the amount of work needed to implement a working system. We model music notation as structured scores that can easily be extracted from existing standards at run-time; we associate to the model an algebra to access to the internal components of the scores; we allow the application of external functions; and finally we integrate the whole design in XQuery, with limited implementation requirements.

We believe that this work brings a simple and promising framework to define a query interface on top of Digital Libraries, with all the advantages of a concise and declarative approach for data management. It also offers several interesting perspectives: automatic content management (split a score in parts, distribute them to digital music stands), advanced content-based search, and finally advanced mining tasks (derivation of features, annotation of scores with these features).

8. REFERENCES

[1] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The active XML project: an overview. *VLDB J.*, 17(5):1019–1040, 2008.

[2] Serge Abiteboul and et al. WebContent: Efficient P2P Warehousing of Web Data. In *VLDB'08 Very Large Data Base*, pages 1428–1431, August 2008.

[3] Mira Balaban. The music structures approach to knowledge representation for music processing. *Computer Music Journal*, 20(2):96–111, 1996.

[4] Michael Scott Cuthbert and Christopher Ariza. Music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data. In *Proceedings of the International Conference on Music Information Retrieval (ISMIR)*, pages 637–642, 2010.

[5] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[6] Dominique Fober, Stéphane Letz, Yann Orlarey, and Frédéric Bevilacqua. Programming Interactive Music Scores with INScore. In *Sound and Music Computing*, pages 185–190, Stockholm, Sweden, July 2013.

[7] Dominique Fober, Yann Orlarey, and Stéphane Letz. Scores level composition based on the guido music notation. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 383–386, 2012.

[8] R. Fournier-S'niehotta, P. Rigaux, and N. Travers. An Algebra for Score Content Manipulation. Technical Report CEDRIC-16-3616, CEDRIC laboratory, CNAM-Paris, France, 2016.

[9] Joachim Ganseman, Paul Scheunders, and Wim D'haes. Using XQuery on MusicXML Databases for Musicological Analysis. In *Proceedings of the International Conference on Music Information Retrieval (ISMIR)*, 2008.

[10] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[11] Michael Good. *MusicXML for Notation and Analysis*, pages 113–124. W. B. Hewlett and E. Selfridge-Field, MIT Press, 2001.

[12] Paul Hudak. *The Haskell School of Music – From Signals to Symphonies*. (Version 2.6), January 2015.

[13] David Huron. Music information processing using the humdrum toolkit: Concepts, examples, and lessons. *Computer Music Journal*, 26(2):11–26, July 2002.

[14] David Janin, Florent Berthaut, Myriam Desainte-Catherine, Yann Orlarey, and Sylvain Salvati. The T-Calculus : towards a structured programing of (musical) time and space. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling and design (FARM'13)*, pages 23–34, 2013.

[15] Music Encoding Initiative. http://music-encoding.org, 2015. Accessed Oct. 2015.

[16] MusicXML. http://www.musicxml.org, 2015. Accessed Oct. 2015.

[17] Philippe Rigaux, Lylia Abrouk, H. Audéon, Nadine Cullot, C. Davy-Rigaux, Zoé Faget, E. Gavignet, David Gross-Amblard, A. Tacaille, and Virginie Thion-Goasdoué. The design and implementation of neuma, a collaborative digital scores library - requirements, architecture, and models. *Int. J. on Digital Libraries*, 12(2-3):73–88, 2012.

[18] Perry Rolland. The Music Encoding Initiative (MEI). In *Proc. Intl. Conf. on Musical Applications Using XML*, pages 55–59, 2002.

[19] Nicolas Travers, Tuyêt Trâm Dang Ngoc, and Tianxiao Liu. Tgv: A tree graph view for modeling untyped xquery. In *12th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 1001–1006. Springer, 2007.

[20] W3C Music Notation Community Group. https://www.w3.org/community/music-notation/, 2015. Last accessed Jan. 2016.

[21] Philip Wheatland. Thoth music learning software, v2.5, Feb 27, 2015. http://www.melodicmatch.com/.

[22] XML Schema. World Wide Web Consortium, 2001. http://www.w3.org/XML/Schema.

[23] XQuery 3.0: An XML Query Language. World Wide Web Consortium, 2007. https://www.w3.org/TR/xquery-30/.