Ask Toscanini!—Architecting a Search Engine for Music Scores Beyond Metadata

Arman Bahraini and Eli Tilevich
Software Innovations Lab
Virginia Tech
Blacksburg, Virginia, USA
{arman1,tilevich}@cs.vt.edu

ABSTRACT

As part of their trade, musicians continuously study and analyze music scores. Important musical tasks such as selecting repertoire require searching scores for their constituent elements. Searching collections of scores can be burdensome and cognitively taxing even for expert musicologists. Despite the promise of computing to facilitate searching over vast volumes of information, commercial search engines only operate on score metadata (e.g., composer, time period, genre, etc.). In this paper, we discuss the user requirements, design choices, and software architecture of a search engine for querying music scores beyond metadata (e.g., instrument range, key/time signature, dynamics, etc.). We also present our proof-of-concept implementation of this architecture—*Ask Toscanini!*—which supports a wide selection of user queries, expressed as structured text strings, against a collection of digital scores. The engine can search through any collection of scores, provided in the popular standardized MusicXML format, which is subsequently transformed into our custom search-efficient format. In addition to ensuring search efficiency, our design also renders the engine amenable for use by musicians, the majority of whom are not computing experts. The insights reported in this paper can help future research efforts in enhancing search technologies for music scores and can serve as a blueprint for creating commercial solutions in this domain.

CCS CONCEPTS

• **Information systems → Search engine architectures and scalability**; • **Applied computing → Sound and music computing**;

KEYWORDS

music scores, information retrieval, search engine, MusicXML, symbolic information

ACM Reference Format:
Arman Bahraini and Eli Tilevich. 2019. *Ask Toscanini!*—Architecting a Search Engine for Music Scores Beyond Metadata. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3297356

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/10.1145/3297280.3297356

1 INTRODUCTION

A music score, also known as sheet music, is a medium through which composers communicate their creative output to performers. Scores are self-contained repositories of musical information, which include tempo[1], time signature[2], key signature[3], instrumentation[4], dynamics[5], etc.—everything required for performers to interpret a musical piece. A medium from which performers play, scores are also commonly searched and studied. In fact, music history reports on some intellectual feats associated with analyzing musical scores.

The great Italian conductor Arturo Toscanini (1867–1957) was returning to his dressing room during the intermission after the first act of a Verdi opera, when the principal oboist nervously burst into his view: "Maestro, I am afraid we have a problem! A key on my oboe just broke, making it impossible for me to play the low B flat." Toscanini thought for a couple of seconds and then reassured his musician: "We should be fine then—there is no B flat in your part in the remaining three acts of the opera." Of course, Toscanini, one of the greatest conductors of the 20$^{\text{th}}$ century, was a musical genius. Not only had he memorized the entire operatic score, but he was also able to scan through the entire oboe part in his mind blazingly fast to come to this reassuring conclusion.

With modern computing technologies, this intellectual feat of Maestro Toscanini should be easily repeatable. Computers should enable musicians to answer many more questions about the content and characteristics of music scores. Unfortunately, major commercial search engines treat music scores as black boxes, operating on score metadata, without the ability to examine the content of the searched scores. Typical score metadata includes the piece's title, composer, time period, style, and sometimes instrumentation. As a result, one can search for all the scores written by Mozart for a string quartet. However, it would be impossible to search for a subset of such scores that, for example, limit the constituent instruments to given pitch ranges.

What if a music search engine could query collections of scores beyond metadata? With a single query, a musician should be able to identify the pieces across the entire orchestral repertoire with instrumental parts in given pitch ranges. A lack of the ability to query scores beyond metadata makes it impossible to ask important questions like that. This limitation constrains our very understanding of the music literature, with only the most knowledgeable musicologists capable of guessing what the answers are. Indeed, getting

[1]The number of beats per minute
[2]The number of beats to a measure expressed numerically
[3]The key as expressed by the ♭ or ♯ symbols at the beginning of a staff line
[4]Arranging music to be played by a given combination of instruments
[5]The level of level of loudness or softness

precise answers even for a single non-trivial piece requires manual analysis, which is tedious and error-prone. Extending this analysis to a collection of scores creates a problem intractable for human experts.

A common real-world use case of manually searching a collection of music scores is finding the repertoire suitable for a particular performance group. Musicians spend hours searching through scores to identify those that would correspond to a given ensemble members' playing abilities [21]. Consider how every school year, thousands of music directors face the problem of finding a repertoire that would fit their ensembles. Unless a score has been analyzed by a musical expert and given a difficulty grade, extant search engines lack the ability to examine each individual instrumental part. Selecting scores of given time and key signatures, with instrument parts within certain pitch ranges, can greatly help determine which scores would be suitable for a given performance group.

However, to properly accommodate its target audience, the engine must offer a query interface that is accessible to musicians. Despite their intimate understanding of music scores, musicians may lack the sophisticated understanding of computing required to interact with a query interface that imposes strict syntactic compliance requirements. We discuss how we overcame this challenge by creating a structured query interface that enables useful search functionality, while tolerating simple syntactic imprecisions.

In this paper, we report on our experiences of architecting a search engine for music scores that can query them beyond metadata. We describe our software architecture and a specific proof-of-concept implementation, called *Ask Toscanini!*. Given a collection of scores in the popular MusicXML format [23], *Ask Toscanini!* accepts structured web-based queries, entered through a search box and specifying various individual elements of the searched scores. In particular, the scores can be searched on time/key signatures, instrument ranges, tempos, and dynamics. Although *Ask Toscanini!* only demonstrates our design ideas, its software architecture provides all the necessary foundations for building commercial quality search engines for music scores.

The rest of this paper is structured as follows. Section 2 discusses the requirements. Section 3 describes the software architecture of our search engine. Section 4 details our reference implementation. Section 5 presents our performance evaluation results. Section 6 compares our approach to the related state of the art. Section 7 presents future work directions and concludes the paper.

2 REQUIREMENTS

To provide value for its target audience, a search engine for music scores must fulfill several requirements. A key challenge in creating a music score search engine that allows querying musical information beyond metadata is overcoming the sharp contrast between the computing sophistication of the userbase and the complexity of the search medium. The engine must be useful for musicians, who possess deep knowledge about musical notation and the fine-grained details of interpreting any given score. Nonetheless, musicians are not expected to possess expertise in computing theory, including the concepts of syntax, logical operators, and data processing. As search media, music scores can be incredibly complex, represented as a multi-layered hierarchy of containers, whose exact structure can be non-trivial to ascertain. Hence, the engine's user interface should abstract the need to operate on non-trivially structured collections of search data to accommodate the target audience.

The user interface must strike the right balance between expressiveness and restrictiveness. It should be able to express multi-conditional queries, while not accepting and returning results to obviously nonsensical requests. If a search query is only partially satisfied due to a semantically invalid condition, the user should not be exposed to wasting their time looking through search results containing unrelated scores. As an example, consider searching for scores, whose tempo ranges between 60 and 80 BPM. The user may mistakenly specify the query as (*tempo 80 and 60*), which can be interpreted either as a reversed range or a simple typo (e.g., *tempo 80 and 160*). Reconciling these requirements raises the need for advanced error reporting. Not only should the engine flag incorrectly constructed queries, but it also should be capable of suggesting meaningful corrections that the users can follow.

The engine must provide high utility, replacing the need for manually processing collections of scores. The accepted queries should be able to filter large collections of scores to produce a reasonably small subset that can be further examined by hand. To that end, queries should be easily composable to enable building increasingly complex queries out of simple individual components. The user should clearly see how the size of the result set decreases in response to adding search conditions.

Finally, spoiled by the spectacular responsiveness of commercial web search engines, modern users expect an almost instantaneous response from emerging search technologies. End users, unaware of the fundamental dissimilarities between the processes of web search and that of searching for scores would expect comparable performance in terms of the response time. At the same time, it would be highly inappropriate to trade precision for speed, like may be permissible in the case of web search. A score incorrectly returned based on a correctly specified query would almost defeat the purpose of a search engine for music scores.

3 SOFTWARE ARCHITECTURE

By fulfilling the requirements outlined above, one can define a software architecture for building engines capable of searching music scores beyond metadata. We first discuss the peculiarities of the MusicXML format, as our engine works with this universal music representation. Then we discuss our architecture, its main components and their interactions.

3.1 MusicXML as a Search Media

Recall that one of our requirements is to provide high utility for an engine operating on a collection of scores. An important question concerns which common format should be used for representing the searched scores. MusicXML is an industry-wide, open-source format supported by numerous major music software makers, including proprietary software and open-source tools [5, 17]. In recognition of the wide embrace of MusicXML, the World Wide Web Consortium (W3C) [19] adopted the development of this format for representing music scores for all web-based purposes and the Library of Congress for digital preservation [1].

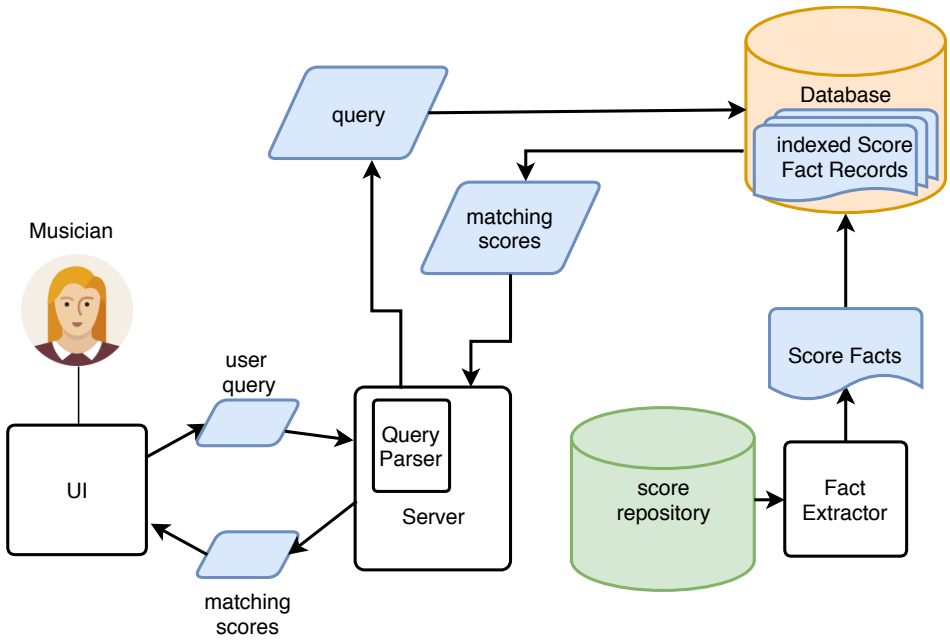

Figure 1: System Architecture for the Search Engine for Music Scores Beyond Metadata.

MusicXML leverages the eXtensible Markup Language (XML) technology [18] in order to provide self-describing tags that express all constituent elements of a music score. MusicXML documents represent hierarchical structures, with the top element of `<score-partwise>`, which serves as the root of a hierarchy of containers. An interesting artifact of MusicXML design is that this universal standard is intended for rendering musical scores in a score editor or playing them on electronic devices [18]. When an editor renders a MusicXML document or a music synthesizer plays it, its content is read sequentially from top to bottom, with the content being processed as it is being parsed and extracted. This sequential access model implies that all search operations on the constituent content of a score would have the linear asymptotic complexity [26] of $O(n)$, where n is the size of a score in terms of the number of nodes in the XML tree. For non-trivial scores in large collections, this linear complexity would yield very inefficient search performance. In addition, the physical constraints on computer memory impose limits on how much data can be searched efficiently. As a result, when searching large volumes of information, search algorithms should possess at least logarithmic efficiency ($O(lg(n))$). Since unprocessed MusicXML documents support only sequential access, we discovered a need to preprocess the searched scores, thereby transforming them into a format that is better amenable to the instantaneous retrieval of solely the important information.

In our reference implementation, we preprocess MusicXML documents into a collection of *score facts*—high level information that pertains directly to user queries such as time signatures, key signatures, etc. We extract and compute score facts from the raw MusicXML documents in a process that we call *fact extraction*. The specific procedures that we follow to extract the relevant facts, as well as their representation, are detailed in Section 4. Further, we discuss the performance improvements that result from operating on preprocessed search media in Section 5.

By querying the extracted facts rather than the original scores, our design reduces the space complexity by orders of magnitude and the time complexity to near constant time. This preprocessing procedure creates the extracted fact documents, structured as a key-value store, known for the nearly constant efficiency of its retrieval operations. Preprocessing is quite computationally intensive, due to the need to parse large XML documents and insert the extracted data to a database for permanent storage. However, the procedure needs to be executed only once when new scores are added from the repository. The resulting costs are then amortized by the sped-up performance for all the subsequent search queries. In commercial deployments, we envision running the fact extraction procedure in batch mode, preferably overnight, when the utilization of the main query interface is at a minimum.

3.2 System Architecture

Figure 1 describes the system architecture that can be used to build search engines for querying music scores beyond metadata. The architecture comprises three main tiers: the client, the server, and the database. In addition, the database tier includes a separate component that preprocesses raw scores into indexed database records; the preprocessing tier operates independently of the main engine functionality. The user enters a search query through the user interface (UI) component and receives back a list of matching scores that can be empty. The Server component processes the queries by parsing them into access requests to the Database component, which in turn returns back the score list, presented to the user as a list of titles, each of which is a link that can be followed to access the actual score.

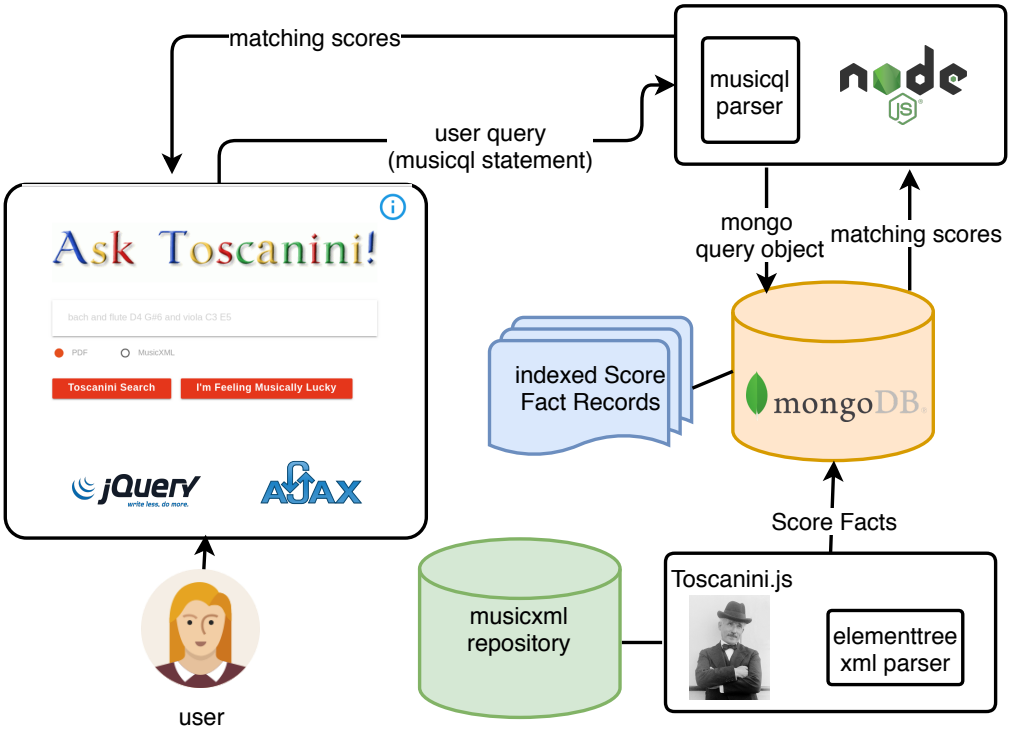

Figure 2: Reference Implementation: *Ask Toscanini!*

In essence, the architecture is structured around the commonly used in enterprise computing Model View Controller (MVC) [22], which provides the benefits of separating key concerns as well as modularizing the implementation to facilitate both forward development and subsequent maintenance. Next we briefly explain how the components described above map into the MVC architecture.

The Model component encapsulates data management, which includes a repository of raw scores and their representations as precomputed score facts. How scores should be transformed into facts depends on the requirements for a given deployment. The transformation process can be performed in batch mode over the entire collection or be performed on demand upon first access. The important consideration is that the results must be cached and persisted for subsequent queries. Another facet to consider is how closely the raw score repository and the facts database are synchronized. For example, should the removal of a raw score from the repository immediately be reflected by the database? Otherwise, a query can return a score fitting the search criteria that is no longer available in the repository.

The View component encapsulates the user interface and its visual representation. There can be a wide variety of approaches to present the front-end of the search engine to the user. An important consideration is whether the search interface should be text-based or use some visual domain-specific language. Text-based interfaces allow for syntax errors that must be handled in a user-friendly fashion. Visual interfaces should make it impossible to enter syntactically incorrect queries, but could necessarily constrain the user. Hence, these two options trade flexibility for correctness, and should be selected based on the deployment requirements.

The Controller component connects the View and Model components. It is responsible for detecting when a query is submitted, passing it to the server for processing, contacting the database, and routing the query results back to the user. As has become a standard usage paradigm for search engines, we advocate following the Request-Reply interaction. In that sense, when a query is in progress, no other queries should be accepted. Although the query aspect of the View component should enforce such sequential operation, the other components should be free to exploit concurrency at will to ensure high performance and responsiveness.

4 REFERENCE IMPLEMENTATION

Next, we reflect on the design decisions and technologies used to implement the aforementioned architecture. We have concretely implemented this architecture as a proof-of-concept search engine, "Ask Toscanini!,"[6] whose main modules appear in Figure 2. For our implementation, we build a web-based system that uses standard enterprise software components, including a Node.js server component and a MongoDB non-relational database engine.

To facilitate portability, we use JavaScript (ES6) as the implementation language throughout the system (both frontend and backend parts). This language's vast third-party library availability and the support for multi-paradigm programming make it the

[6] *Ask Toscanini!* is hosted at toscanini.cs.vt.edu

lingua franca for prototyping any web-based systems for the enterprise. For commercial deployments, some of the system's modules can be reimplemented in other languages, but in our evaluation, we have not encountered any performance bottlenecks that would make us question our choice of implementation language. In our implementation practices, we emphasize the functional programming aspects of JavaScript with its support for data immutability, function composition and reuse. However, our adherence to the functional programming paradigm is not exclusive. We also make use of object oriented features (for abstracting processes) and procedural interfaces (for accessing persistent data).

Next, we detail the implementation insights of the key modules of our reference implementation, to explain our design choices and how they were shaped by the constraints of the constituent technologies. In the following subsections, the titles include the module name and its description under which it appears in Figure 2—i.e., Module ("Module Name").

4.1 Score Repository ("MusicXML Repository")

MusicXML contains extraneous information, relevant only for rendering and playback. Our fact extractor (discussed in 4.2) filters out the irrelevant information.

4.2 Fact Extractor ("Toscanini.js")

Before we discuss how we extract the relevant facts from a score, consider Figure 3, which shows the extracted facts from a Bach cantata in MusicXML[7]. In essence, facts are a collection of answers to the supported queries, formatted as a JSON [9] document. Because JSON documents can be searched in constant time, the precomputed facts are retrieved instantaneously. Besides, the size of a typical facts document is orders of magnitude smaller than the score file it represents. A typical facts document lists the pitch ranges for all the constituent instruments, tempo ranges, key signatures, time signatures, and dynamics. As the search engine's capacities grow, the facts format needs to evolve accordingly to include the information required to answer newly supported queries.

For the reference implementation, we have created a fully functional open source fact extractor, *Toscanini.js*[8]. The module can be instantiated as an object, taking as input a single MusicXML string representing the entire score. Individual score elements can be retrieved by calling methods, such as `getPitchRange(instrumentName)`, which retrieves the playing range for a given instrument as a JavaScript object (dictionary)—`{``minPitch'': 30, ``maxPitch'': 72}`.

The fact extraction process includes the following phases: **Phase 0:** Check if the score repository has new MusicXML scores since the last fact extraction, observable if the repository has score file names not recorded in the database. **Phase I:** Parse the MusicXML of a new score into a programmatically searchable data structure. **Phase II:** Search for relevant XML elements and compute facts. For example, to find the playing pitch ranges for the instruments, the MusicXML `<pitch>` elements should be analyzed. **Phase III:** Repeat Phase I and II for each new score. **Phase IV:** Insert the new score facts into the database. **Phase V:** Index the database based on score title and the extracted facts.

Figure 3: Extracted Facts Document for "Was mein Gott will, das g'scheh allzeit," BWV 111, by J.S. Bach


```
{
  "_id": "bach_wasmeingott.xml",
  "instrumentRanges": [
    {
      "minPitch": 54,
      "maxPitch": 64,
      "instrumentName": "soprano"
    },
  ...
    {
      "minPitch": 33,
      "maxPitch": 47,
      "instrumentName": "bass"
    }
  ],
  "minTempo": 96,
  "maxTempo": 96,
  "keySignatures": [
    "d"
  ],
  "timeSignatures": [
    {
      "beats": 4,
      "beatType": 4
    }
  ],
  "dynamics": []
}
```


Note: pitch is stored as MIDI numbers for range queries.

Under the hood, Toscanini.js utilizes the popular *elementtree* XML parser module and its support for XPATH expressions, such as *//dynamics* to retrieve all dynamics tags in the score irrespective of their relative position in the score document. Elementtree also offers ease of deployment, as it requires no compilation or complex configuration, while offering acceptable performance characteristics. We have also experimented with event-based parsing strategies, such as the SAX-based XML parser, xml2js; however, we found its functionality lacking in not being able to easily determine the relative position of MusicXML elements, an essential functionality for our fact extraction process.

4.3 Database ("MongoDB")

In modern database theory, there is serious debate about the range of applicability of relational (i.e., SQL-based) vs. NoSQL databases. Although relational databases have been dominant in enterprise computing, emerging domains make increasing use of non-relational databases [13]. When designing the conceptual data model for representing the score facts database, we discovered that the underlying data representation is not inherently relational—the score facts can be naturally represented as self-contained documents, as appears in Figure 3. In contrast, a relational representation would lead to the creation of numerous entities (i.e., tables). For example, instrument specific information such as pitch range, would need to be represented in separate tables, with each range value appearing in its own table entry. As a result, performing the most rudimentary queries would require the execution of expensive multi-table operations (i.e., joins), known to impose a heavy performance overhead.

[7] https://drive.google.com/file/d/1Gjg3rcKnj-UOD_xjRr2qVCR18KoVDq9V/view?usp=sharing.
[8] https://www.npmjs.com/package/toscanini

An additional requirement that needs to be fulfilled is keeping the data model open for ease of evolution, as new types of queries become necessary to support.

To address our aforementioned observations, our reference implementation makes use of the popular document-oriented database, MongoDB. MongoDB stores score data as JSON documents, while providing all facilities one expects from a traditional database system, including ad-hoc querying, aggregation, data persistence, caching, and JavaScript integration. Not only does MongoDB store JSON documents, but it also accepts JSON-formatted queries, thus facilitating a consistent programming interface.

4.4 Parser ("PEG.js - musicQL")

Our reference implementation uses a flexible text-based query interface. This flexibility, however, comes at a steep price of the user being able to enter any string of text, potentially erroneous or even nonsensical. Our design goal is to help the user correct their queries by giving them helpful hints. To that end, *Ask Toscanini!* features a query parser that checks the submitted queries against a formal grammar, while reporting back the grammar violations as helpful correction suggestions. By iteratively correcting the reported errors, users can evolve their inchoate queries into sophisticated interactions with the search engine. Upon validating a query, our parser produces a query object for MongoDB to search the fact documents.

We call our query parser and associated grammar rules "musicQL." It accepts structured queries (e.g., *beethoven and dynamic mf and tempo 75 150*) and checks them against a grammar. The aforementioned example query specifies a composer name (`beethoven`), a dynamic (`mf`), and a tempo range (`75 150`), respectively. Figure 4 shows the output of this query's execution—only one score was returned, Beethoven's Symphony No.4 Movement 4, whose PDF appears as a hyperlink.

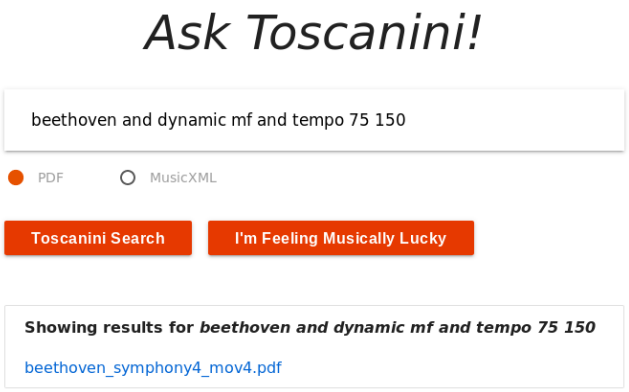

Figure 4: The search engine in action

Other example queries include:

- See all scores we have by leaving the search box empty
- Query for an instrument with a specific range, ex: trumpet F#3 D5
- Query for composer or instrument (no range), ex: "bach", "viola", etc.
- Query for tempo between range, ex: "tempo 40 130"
- Query for key signature, ex: "key Bb"
- Query for time signature, ex: "ts 2 4"
- Query for dynamic (anywhere in the score), ex: "dynamic mf". Currently the supported representations for dynamics are: f, ff, fff, ffff, fffff, ffffff, fp, fz, mf, mp, p, pp, ppp, pppp, ppppp, pppppp, rf, rfz, sf, sffz, sfp, sfpp, sfz
- Putting it all together: "Beethoven and tempo 75 150 and flute D4 G#6 and trumpet F#3 A5 and key Bb and ts 2 4 and dynamic mp and dynamic ff"

Our parser was generated to check the syntactic correctness of the input queries based on a formal grammar [14]. The formal grammar of our parser extends the classic EBNF format into the Parsing Expression Grammars ("PEGs") model [12], which offers superior parsing performance and ambiguity prevention . PEGs fit our problem domain well in their ability to eliminate the ambiguity of mistaking similar queries for each other.

```
start = _(clause)(_"and"_ clause)*_

clause = (musicTerm / instrumentRange / composerInstrument)

musicTerm = "ts"_ beats:([1-9][0-9]?) _ beatType:([1-9][0-9]?)
/ "tempo" _ min:([0-9][0-9]?[0-9]?) _ max:([1-9][0-9]?[0-9]?)
/ "key" _ key:([a-gA-G][b|#]?)
/ "dynamic" _ dynamic:
("ffffff"/"fffff"/"ffff"/"fff"/"ff"/"fp"/"fz"/"f"/"mf"/
"mp"/"pppppp"/"ppppp"/"pppp"/"ppp"/"pp"/"p"/"rfz"/
"rf"/"sfz"/"sffz"/"sfpp"/"sfp"/"sf")

instrumentRange = instrument:([a-zA-Z0-9])+
_ min:([a-gA-G][b|#]?[0-9]) _ max:([a-gA-G][b|#]?[0-9])

composerInstrument = ci:([a-zA-Z0-9]+)

_ "whitespace" = [ \t\n\r]*
```

Figure 5: musicQL's Parsing Expression Grammar

For brevity, we show the musicQL grammar, but elide the query object creation in Figure 5. The grammar rules are *start, clause, musicTerm, instrumentRange,* and *composerInstrument*. We leverage PEGs prioritized choice operator, the "/", which picks the first successful match. For example, by ordering the supported dynamics carefully, the query `dynamic ff` could never be mistaken for `dynamic f`.

Figure 6: Error message in response to an invalid query

Our parser aims to be semantically strict, but syntactically forgiving. We want to save the time wasted on examining scores returned in response to a partially satisfied query with an invalid condition. Consider the query *beethoven and dynamic m and tempo 75 150*.

This query would not pass our grammar because "m" is not a valid dynamic, so the user would see the error message in Figure 6.

4.5 Server ("Node.js")

Node.js https://nodejs.org/en/about/ is the de facto framework for developing server functionality in JavaScript. Its programing model naturally supports efficient asynchronous input/output operations, such as reading from disk, an important feature for a search engine. Node.js offers superior performance, being powered by Google's highly optimized JavaScript engine V8. Due to the portability of Node.js, our reference implementation can be deployed in many operating environments, as long as they follow standard deployment conventions. We have seamlessly deployed *Ask Toscanini!* in the Windows, OSX, Fedora, and CentOS operating environments.

4.6 UI ("Ajax, JQuery, HTML, Materialize-CSS")

Our user interface is offered through the standard browser-based front end. In our implementation, we follow the industry standards, which incorporate some of the most tried and tested technologies. All HTTP requests to the server use Ajax to avoid blocking the browser UI thread. JQuery—the "write less, do more" JavaScript library—handles all the interactions with the HTML DOM and issues the Ajax requests in a structured fashion. We use the Materialize styling framework (https://materializecss.com/about.html), which is based on principles set forth by Google to achieve consistent look and feel across devices. Finally, we leverage CSS's `@media` functionality [8] for fine-tuned control of styling across screen sizes. Since our target audience may access the search engine from a variety of computing devices, our user interface can flexibly adapt to dissimilar devices.

5 PERFORMANCE EVALUATION

The goal of our performance evaluation is to validate the need for extracting facts. To that end, we first benchmarked the latency of executing all supported queries against the entire collection, comprising of 35 classical pieces of different sizes and orchestrations taking about 40MB of disk space and representing over 500 pages of sheet music. By comparison, the extracted facts for these scores take about 20KB of disk space to store. Our system configuration for these benchmarks is: Macmini5.3, Intel Core i7, 2 GHz, 4 cores, 8 GB RAM. Each benchmark was repeated 10 times, with the results averaged. Figure 7 shows the breakdown of the total time taken by (1) reading scores from the disk (.32 sec), (2) parsing MusicXML (33.12 sec), and (3) searching the parsed documents (1.02 sec). As one can see, parsing dominates the processing costs, with searching being a far second. Third-party benchmarks suggest the actual performance of an XML parser bears only a minor impact on the overall system performance [2].

The ability to pre-parse MusicXML files has undeniable performance improvement benefits. However, as we discovered, searching extracted facts offers additional performance improvements. To quantify these additional improvements, consider Figure 8 that compares the time it takes to search pre-parsed MusicXML vs. extracted facts. The obtained performance numbers shows that the extracted facts can be searched in a fraction of the time it takes to search even pre-parsed MusicXML documents.

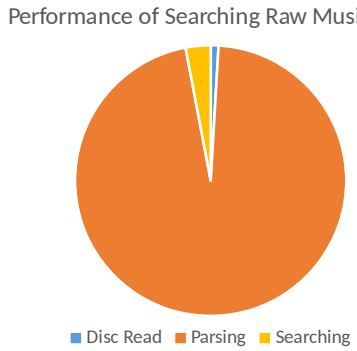

Figure 7: Breakdown between constituent parts of a query

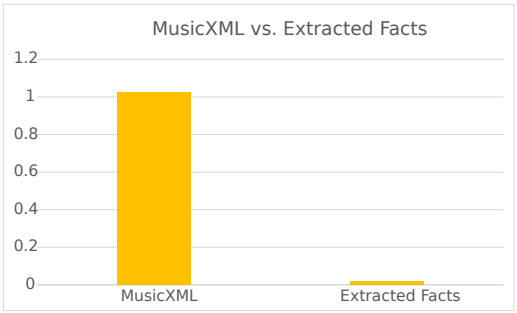

Figure 8: Searching Cached MusicXML vs. Extracted Facts

6 RELATED WORK

One of the distinguishing features of *Ask Toscanini!* is making use of a database engine to systematically organize and efficiently retrieve score-related information in order to process user queries. Our approach follows on an earlier insight about the need for database support as a means of maintaining collections of symbolic music information [20]. Furthermore, our requirements are molded by user expectations stemming from interacting with hypertext search engines, which emphasize result correctness [10]. These general-purpose search engines, however, cannot search music scores beyond metadata. Another feature of hypertext engines we consider adopting is ranking the search results to supplement the current exact matches that satisfy the user-specified search conditions.

To process queries, a search engine must first parse and extract information from music scores in a digital format. General libraries, frameworks, and toolkits come very handy in this context. For example, Music21 [11] is a very powerful digital musicology toolkit for Python programmers. In fact, this toolkit has already been used to extract some musical content [16]. Although we could have leveraged Music21 in our reference implementation, we instead decided to develop our own score processing toolchain in JavaScript to be specifically tailored toward searching. In our implementation, the *Toscanini.js* module provides the necessary functionality exposed through a simple API for fact extraction.

One of our key design choices was using a simple text-based query interface, through which users without any computing background can enter useful queries against collections of scores. Our

architecture provides for a rigorous and systematic checking of the syntactic validity of the entered queries, giving end-users helpful correction suggestions. Prior related work has advocated the use of the XQuery technology for extracting information from XML documents [15, 25]. Indeed, XQuery is highly powerful and expressive for its intended usage domain—querying XML documents in enterprise applications. However, using XQuery in the context of a search engine for scores would be inappropriate for two reasons. First, it would be highly confusing to expose MusicXML as the actual search media to the end user. Our approach uses this formate strictly for internal processing, with the end user designing their queries based on actual musical content rather than its MusicXML representation. Second, XQuery requires certain programming expertise to use correctly. Requiring this expertise would introduce an unnecessary obstacle to adoption for our target audience. Finally, these prior approaches left the questions of the performance and scalability of applying XQuery unaddressed. In contrast, our benchmarks validate our architectural and implementation choices.

Liber Usualis [6] bares some similarity with *Ask Toscanini!* in its support for content based feature querying, such as pitch sequence and text search. Although relevant for musicologists, these queries would be unhelpful for a band director or performer trying to find repertoire to play. The project provides no support for combining multiple queries, using a single drop down menu UI. We plan to incorporate that engine's ability to observe where in a score a certain query is answered. For example, a pitch sequence query will show the measures where said pitch sequence occurs.

As compared to retrieving scores based on a search criteria, other related approaches enable statistical inferencing on a collection of scores. Given a collection of jazz charts, the user can make statistical inferences based on the probability of the presence of certain chords and individual notes [24]. In our future work, we may consider adding support for queries based on statistical properties of the searched scores.

7 FUTURE WORK AND CONCLUSIONS

As an ongoing research project, our reference implementation is being continuously evolved, optimized, and evaluated. In particular, we are innovating in the end-user interface space and error correction suggestions to better accommodate our target audience. We are also working on supporting querying for other elements of structure and content. Handing truly large collections would require high scalability, with the engine running in parallel on a compute cluster, a direction that we plan to explore. Fortunately, MongoDB naturally supports such data parallelism through sharding and map-reduce [3, 4, 7]. In addition to searching, parallel processing can also speed-up our fact extraction process.

We have presented *Ask Toscanini!* by reflecting on our experiences of architecting, designing, and implementing a search engine for music scores beyond metadata. Key insights that we gained include the need to preprocess MusicXML documents to ensure high search performance as well as the necessity of systematic error checking and correction for user queries to accommodate our target audience. The design ideas discussed herein and our preliminary evaluation results can help drive efforts in creating powerful automated exploration technologies for music scores, empowering both musicians and technologists.

ACKNOWLEDGEMENTS

The *Ask Toscanini!* project has been made possible in part by a grant from Virginia Tech's Institute for Creativity, Arts, and Technology (ICAT). The co-authors would like to acknowledge Dr. Charles Nichols for his interest in and support for this research. Spencer Lee as well as undergraduate CS and Music researchers—Galina Belolipetski, Taber Fisher, Matthew Fishman, and Michael Mills—have served as a patient and enthusiastic audience for this research.

REFERENCES

[1] 2012. MusicXML, Version 3. http://www.digitalpreservation.gov:8081/formats/fdd/fdd000358.shtml [Online; accessed 24. Jun. 2018].
[2] 2018. astro/node-expat. https://github.com/astro/node-expat [Online; accessed 24. Jun. 2018].
[3] 2018. Map-Reduce and Sharded Collections — MongoDB Manual. https://docs.mongodb.com/manual/core/map-reduce-sharded-collections [Online; accessed 24. Jun. 2018].
[4] 2018. Map-Reduce — MongoDB Manual. https://docs.mongodb.com/manual/core/map-reduce [Online; accessed 24. Jun. 2018].
[5] 2018. MusicXML for Exchanging Digital Sheet Music. https://www.musicxml.com [Online; accessed 24. Jun. 2018].
[6] 2018. Search the Liber Usualis. http://liber.simssa.ca [Online; accessed 5. Sep. 2018].
[7] 2018. Sharding — MongoDB Manual. https://docs.mongodb.com/manual/sharding [Online; accessed 24. Jun. 2018].
[8] 2018. Using media queries. https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries [Online; accessed 24. Jun. 2018].
[9] Tim Bray. 2017. *The JavaScript object notation (JSON) data interchange format (No. RFC 8259).* Technical Report.
[10] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* 30, 1-7 (1998), 107–117.
[11] Michael Scott Cuthbert and Christopher Ariza. 2010. music21: A toolkit for computer-aided musicology and symbolic music data. (2010).
[12] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 111–122.
[13] Marin Fotache and Dragos Cogean. 2013. NoSQL and SQL databases for mobile applications. Case study: MongoDB versus PostgreSQL. *Informatica Economica* 17, 2 (2013), 41.
[14] Futago-za Ryuu (futagoza. ryuu@gmail. com). 2018. PEG.js – Parser Generator for JavaScript. https://pegjs.org [Online; accessed 24. Jun. 2018].
[15] Joachim Ganseman, Paul Scheunders, and Wim D'haes. 2008. Using XQuery on MusicXML Databases for Musicological Analysis. In *ISMIR*. 433–438.
[16] David Garfinkle, Claire Arthur, Peter Schubert, Julie Cumming, and Ichiro Fujinaga. 2017. PatternFinder: Content-Based Music Retrieval with music21. In *Proceedings of the 4th ACM Int. Workshop on Digital Libraries for Musicology*. 5–8.
[17] Michael Good. 2001. MusicXML for notation and analysis. *The virtual score: representation, retrieval, restoration* 12 (2001), 113–124.
[18] Michael Good. 2002. MusicXML in practice: Issues in translation and analysis. In *Proceedings of the First International Conference MAX*. 47–54.
[19] Michael Good. 2017. MusicXML. https://www.w3.org/2017/12/musicxml31/
[20] Goffredo Haus and Maurizio Longari. 2001. Music information description by mark-up languages within DB-Web applications. In *Web Delivering of Music, 2001. Proceedings. First International Conference on.* IEEE, 71–78.
[21] Ethan Holder, Eli Tilevich, and Amy Gillick. 2015. Musiplectics: Computational Assessment of the Complexity of Music Scores. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2015)*. ACM, New York, NY, USA, 107–120.
[22] Avraham Leff and James Rayfield. 2001. Web-application development using the model/view/controller design pattern. In *Proceedings of the 5th IEEE International Enterprise Distributed Object Computing Conference (EDOC'01)*. IEEE, 118–127.
[23] Makemusic Inc. 2015. MusicXML. http://www.musicxml.com/.
[24] François Pachet, Jeff Suzda, and Dani Martinez. 2013. A Comprehensive Online Database of Machine-Readable Lead-Sheets for Jazz Standards. In *ISMIR*. 275–280.
[25] Raffaele Vigliante. 2007. MusicXML: An XML based approach to automatic musicological analysis. *Proc. Digital Humanities* (2007), 4–8.
[26] Weixiong Zhang. 1999. *State-space search: Algorithms, complexity, extensions, and applications.* Springer Science & Business Media.