MusicSQL ReadMe File

Bret Aarden

April 23, 2007

Contents

WARNING: This is alpha-release software! Future releases are subject to change, including re-designs of the database structure which will require deleting old databases and re-importing data.

For a general introduction to the concepts behind MusicSQL, you may want to skip ahead to the section showing "Some example queries".

1 Technical overview

MusicSQL is designed as a 3-tier system for conducting searches of symbolic music databases. The three tiers are:

- Tier I: RDBMS (relational database management system)
- Tier II: ORM (object-relational mapper)
- Tier III: GUI (graphical user interface) [UNIMPLEMENTED]

This ReadMe file provides information on using the ORM, the second tier of MusicSQL. Instead of working directly with the SQL language used by Tier I, the ORM provides a simple Python programming interface. Python is a scripting language that is both easy to learn and object-oriented (see `http://www.python.org`). MusicSQL users need to learn only minimal amounts of Python to construct queries.

In the future MusicSQL will have one or more GUIs which will allow users to build queries visually. Until then, the process must still be scripted.

2 Usage

The "`musicsql`" script provides a basic interface to the MusicSQL libraries and database backends. To get usage information about musicsql, type '`musicsql`' in a Terminal (MacOS) or Command Prompt (Windows) and hit Enter. This will show you a list of available commands. Each musicsql command has its own usage information that can be accessed by typing '`musicsql command`' and hitting Enter. (Replace 'command' with the name of the command you are interested in.)

To get started, use the '`musicsql setup`' command to create a new database. This command also requires a "`--database`" argument specifying the name of your new database. (As a shortcut, instead of typing out long arguments that begin with two dashes, you can just type one dash and the first letter of the argument.) MusicSQL assumes that you are running the free MySQL database server (see "Installing MySQL", below).

To import data from a MusicXML file, use the '`musicsql import`' command. MusicXML is the standard music notation interchange format. It can be exported from notation packages like Finale and Sibelius, music OCR software like SharpEye and SmartScore, and many others. You can find a list of compatible software at this website:

`http://www.recordare.com/xml/software.html`

MusicXML scores are available online at websites like Project Gutenberg and the Choral Public Domain Library. They can also be converted from Humdrum format using "hum2xml". Currently only MusicXML 1.1 is supported.

`http://www.gutenberg.org/browse/categories/4`
`http://www.cpdl.org`

http://extras.humdrum.net/man/hum2xml

To create a PNG or PDF version of a MusicXML file, use the 'musicsql previewxml' command. This makes use of LilyPond (http://www.lilypond.org), which comes bundled with the combo installer. (At present previews are generated on A6 paper size, and PNG previews include only the first staff. You can also use any other MusicXML-compatible tools to generate visual notation.)

If you construct a MusicSQL query using the 'exportable' option (see "Querying in detail", below), the output will include extra columns that can be used to construct preview images of the search results. Use the 'musicsql preview' command to generate previews from these results.

Other musicsql functions are described in the "Querying in detail" section, below.

3 Installing MySQL

MusicSQL assumes that you have MySQL running on your computer. MySQL is a full-fledged database server that runs as a background process. If you don't already have MySQL installed, instructions are provided below.

> It is also possible to use PostgreSQL or SQLite servers instead of MySQL. PostgreSQL has comparable performance to MySQL, but is somewhat harder to install. SQLite can be easy to install, but its limited ability to optimize queries makes it unusable for many tasks. To select one of these other servers, use musicsql's '--backend' option followed by 'postgres' or 'sqlite'.

The easiest way to install MySQL on the MacOS or Windows is to download the installer package from the MySQL website:

http://dev.mysql.com/downloads/mysql/5.0.html#macosx-dmg
http://dev.mysql.com/downloads/mysql/5.0.html#win32

Run the main installer. On Windows, choose the standard configuration, choose to install it as a windows service, and don't modify the security settings. You will probably want to tell the installer to start MySQL on startup automatically.

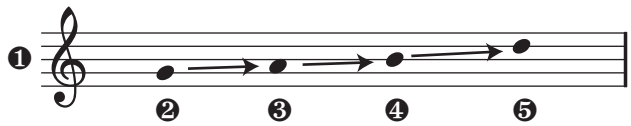
On the MacOS, just follow the directions and the server will be installed. Then double-click on the MySQL.prefPane next to the installer and you will be prompted to install the pane in System Preferences. Now to start the server, simply click the "Start" button in the new MySQL preference pane. If you want the server to start automatically when you start your computer, select that option in the preferences.

If you want to directly access data in MySQL, you may wish to download a standalone client such as the MySQL GUI Tools:

http://dev.mysql.com/downloads/gui-tools/5.0.html

4 Some example queries

Suppose we wanted to search for the ascending pitch sequence pictured below: G5, A5, B5, D5. The steps for constructing this query are explained below, and the actual Python code is shown after each step. In this search we are not looking for particular durations, but there is no reason why we couldn't.



1. Every MusicSQL query starts with a Part which is given to you. In the diagram the Part is pictured as a staff. Even though it is shown with a G clef, that is only shown for convenience — there is nothing in the query pattern so far which specifies a clef. (Unless the clef matters to your search, most often you will not specify one.) The first line of code usually looks like this:

```
part1 = SQLsetup(**options)
```

2. Each object can generate other musical objects, and queries are built piece by piece from existing objects. First we ask the Part to give us a Note on G5. At each step we assign the new object to a variable name so that we can use it again in later steps.

```
note1 = part1.add_note('G5')
```

3. We then ask the Note to give us its next Note on A5.

```
note2 = note1.add_next_note('A5')
```

4. We then ask that Note to give us its next Note on B5.

```
note3 = note2.add_next_note('B5')
```

5. Finally we ask that Note to give us its next Note on D6.

```
note4 = note3.add_next_note('D6')
```

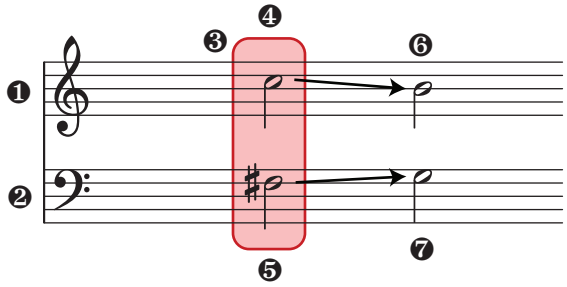
6. We can convert the current query to a usable query at any time.

```
sql = assemble_query()
```

There are also convenience methods for constructing melodies. Once we have a note, we can use it to generate a sequence of notes all at once.

```
part1 = SQLsetup(**options)
note1 = part1.add_note('G5')
note2, note3, note4 = note1.add_note_sequence('A5',
  'B5', 'D6')
```

Here's another example, this one in polyphony:



1. As before, the first step is to get the first Part.

```
part1 = SQLsetup()
```

2. Any number of Parts can be generated separately; they are the only objects that can be self-generated.

```
part2 = Part()
```

3. In MusicSQL, things that happen at the same time are said to start on the same Moment. Now we ask the Part to generate a Moment. In the figure above the Moment is represented as a colored box.

```
moment = part1.add_moment()
```

4. In the last example we used a Part to create a Note. That was a special circumstance: while our query is empty, a Part can generate a single Note or a single Moment out of nothing, but from then on Parts can only create Notes in coordination with Moments. (In point of fact, in the last example the Part first created a Moment behind the scenes and then added a Note to it.) To create a Note, we ask the Moment for a Note using the first Part. Furthermore, we request a half note on C6. The notation puts the duration first, followed by the pitch information.

```
note1 = moment.add_note(part1, '2C6')
```

5. Now that we have the Moment, we can use it to create a Note in the other Part as well. We specify this Note should be a half note on F#4.

```
note2 = moment.add_note(part2, '2F#4')
```

6. We then ask the Note in the first Part to add the Note that follows it, a half note on B5.

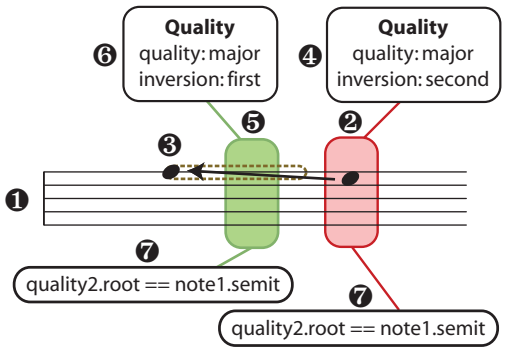
```
note1b = note1.add_next_note('2B5')
```

7. And then we ask the Note in the second Part to add the Note that follows it, a half note on C5.

```
note2b = note2.add_next_note('2G4')
```

Finally, let's explore a more complicated query. This time we'll search for Neapolitan chords that resolve to second-inversion tonic chords. Neapolitans are major triads built on the lowered scale degree 2 (e.g., D-flat in the key of C major) that resolve to a dominant chord, most often a cadential six-four chord that looks like a second-inversion tonic chord.

The typical voice-leading for this progression has the Neapolitan in first inversion, and the root of the Neapolitan in an upper voice resolving down by step to the tonic of the scale. A diagram of this arrangement is shown below, followed by one possible list of steps for constructing it in MusicSQL.



1. Get the first Part.

```
part1 = SQLsetup(**options)
```

2. Add a Note to the Part.

```
note2 = part1.add_note()
```

3. Add the Note before, and specify that it moves down by one chromatic step. This can be specified using the method's "`chromatic_interval=`" named argument.

```
note1 = note2.add_previous_note(chromatic_interval=-1)
```

4a. Now that the basic scaffolding is built, we can start adding other constraints. For instance, the quality of the chord starting on Note 2 should be major. MusicSQL databases don't automatically contain information about chord quality, but we can attach additional information nodes to the database. To add quality information, type "`musicsql add quality`" in the Terminal. (You can get a list of all the currently available replace "add" modules with the "`musicsql list modules`" command.) The command "`musicsql list nodes`" will show you that the new "quality" node is attached to a Moment. We can get this "quality" node via the Moment's "`node`" method.

```
quality2 = moment.node('quality')
```

4b. We want to specify that the Moment's quality should be Major in 2nd inversion. To find out what values are available for a property, use the "`musicsql list values node property`" command (e.g., "`musicsql list values quality inversion`").

```
moment.add_constraint(quality2.c.quality == 'major')
moment.add_constraint(quality2.c.inversion ==
  'second')
```

5a. We want to specify what harmony leads into Note 2. But in real music we can't assume that the harmony occurs right when Note 1 starts. Instead, what we really want to specify is that the harmony occurs *at some point* during Note 1. MusicSQL can represent this situation using a Simultaneity. A Simultaneity attached to one Note will match any Moment that occurs during the length of the Note. If multiple Notes are connected to a Simultaneity, it will match any Moment during which all the notes overlap. Simultaneities are otherwise just like Moments, and share all of their properties and methods. We first add a Simultaneity to Note 1 and obtain its 'quality' node:

```
simul = note1.add_simultaneity
quality1 = simul.node('quality')
```

5b. Next we specify that the quality of the Simultaneity should be a Major chord in 1st inversion:

```
simul.add_constraint(quality1.c.quality == 'major')
simul.add_constraint(quality1.c.inversion == 'first')
```

6. Finally, we set both Notes to be the roots of their chords. We can do this using the "root" property of the "quality" node, which is a PC value. To calculate the PC of the Note, we take its semitone value ("semit") modulo 12:

```
note1.set_constraint(note1.c.semit % 12 ==
  quality1.c.root)
note2.set_constraint(note2.c.semit % 12 ==
  quality2.c.root)
```

These are the general outlines of creating searches. When starting a search, it is helpful to visually sketch out the desired search pattern, and then construct the query using the sketch as a reference.

A few more details are needed to actually run these queries and extract information from them, however. The next section fills in these gaps.

5 Querying in detail

The command "`musicsql template`" will output a blank template for constructing basic MusicSQL queries. This is a short Python script that you can edit to make your own query. To use it, save this template to a text file and open it in an editor. Make sure when you save changes that you keep it in text file format, and not in some other format like Word or RTF. For example:

```
musicsql template > my_query.py
```

To run your query in a Terminal or Command Prompt window, set the current directory to the folder where your query script is located. Then type, for instance, "`python my_query.py`" and hit Enter. (On the MacOS you can invoke the script directly if it has executable permissions). The empty template should start and finish without producing any results.

You will need to specify the database you want to query. To list the databases that are available on a MySQL server, use the "`musicsql list databases`" command.

5.1 MusicSQL objects

The MusicSQL query objects that should be familiar to musicians are Parts and Notes. The other units, perhaps less familiar, are Moments, Events, Noteheads, and Simultaneities. "Moments" represent tiny slices of time and are the means of representing time in a query. "Events" represent the intersections of Parts and Moments. You could imagine Parts as staves on a score, and a Moment as a line drawn vertically through all the staves in a system. A Note that appears on that line will start on the Event that occurs on the intersection between its Part and that Moment.

"Noteheads" encode note information as it appears in a MusicXML file or on a score. A "Note" represents one or more Noteheads tied together: in effect it represents the true start and end information for a pitch. Noteheads represent pitches as they appear on the score even for transposing instruments, but Notes represent sounds at concert pitch. Most of the time you will want to construct queries using Notes, but you can also search the Noteheads associated with Notes.

Music notation focuses mainly on note onsets, but it is often useful to search for Moments that occur at any time over the duration of a Note. A "Simultaneity" is a special type of Moment that is not fixed in time, but can occur at any time that the Notes it is connected to are sounding.

5.2 Building a scaffolding

One useful way to construct your query is to first build a "scaffold" of MusicSQL objects, and then later add constraints and relationships among objects.

The template script starts by creating a single Part, which it assigns to the variable "part1". A new Part can then be created separately. For instance:

```
part2 = Part()
```

Any number of Parts can be created in this manner. Every other object must be generated from other objects, however. In this way you construct a scaffolding of relationships, and every new object must be built off the existing scaffolding. Only Parts can start new structures.

Each object has methods that can be used to add new objects. The currently available methods are listed below:

- Notehead:
  - add_next_tied_notehead()
- Note:
  - notehead()
  - add_simultaneity()
  - add_chordnote()
  - add_previous_note(*details, diatonic_interval*)
  - add_next_note(*details, diatonic_interval*)
  - add_previous_beat(*details, diatonic_interval*)
  - add_next_beat(*details, diatonic_interval*)
  - add_previous_semit(*details, diatonic_interval, contour*)
  - add_next_semit(*details, diatonic_interval, contour*)
  - add_previous_note_or_beat(*details, diatonic_interval*)
  - add_next_note_or_beat(*details, diatonic_interval*)

- Moment:
  - add_note(Part, *details*)
  - add_later_moment(*measure_offset*)
  - add_earlier_moment(*measure_offset*)
  - add_event(Part)
- Part:
  - add_note(*details*) [only before a Moment or Note is added]
  - add_moment() [only before a Moment or Note is added]
  - add_note(*details,* moment=Moment)
  - add_event(Moment)
- Simultaneity:
  - add_note(*pitch*)
  - attach_note(Note)

Note that arguments in italics are optional. Also note that the "`add_note`" and "`add_moment`" methods of a Part can only be used if no Notes or Moments have yet been created.

When a Note is created you can specify details of its pitch or duration. Duration is stated in numbers and dots (e.g., '16.' means dotted-sixteenth note, and '2..' means double-dotted half note). Pitch or pitch class is stated in ISO pitch format, using "#" to represent a sharp and '-' for a flat, '##' for a double-sharp, '–' for a double-flat, and so forth (e.g., 'A#5'). Octave can be omitted if you prefer to search for pitch classes.

When constructing a Note from another Note, you can opt to specify its relative pitch in diatonic steps (use the named argument "`diatonic_interval=`"). For instance, using this method any kind of A is '1' away from any B in the same octave, regardless of accidentals.

The "`add_next_semit`" and "`add_previous_semit`" methods also support searching by contour. Using the named argument "`contour=`", you can specify the pitch direction using the symbol '^' (carat) to indicate a rise, and '`v`' to indicate a fall.

Notes also have three convenience methods: "`add_note_sequence`", "`add_diatonic_interval_sequence`", and "`add_contour_sequence`". Give the method a list in the formats specified above, and it will return a list of notes.

When you create a Moment from another Moment, you can specify the "`measure_offset`", or how many measures separate the two Moments. A distance of 0 means they are in the same measure. If you do not specify the "`measure_offset`" the distance between them is only constrained by other

relationships in the query.

Objects also have features that refer to the objects they are connected to. The following table lists the object connections you can refer to:

```
Event.part
Event.moment
Note.start_event
Note.onset_notehead
Notehead.note
Notehead.tied_previous
```

Most users will probably assume that the "measures" and "attributes" nodes are attached to Moments, but in fact they are attached to Events. (Events are the objects that occur at the intersections between Moments and Parts. Their "attributes" include information about clefs, key signatures, meters, etc.) MusicXML is structured so that each Part has its own independent measure and attribute information, and this same structure is built into MusicSQL.

In order to make "measures" and "attributes" accessible to these users, Moments have methods for accessing these nodes directly. These methods accept an option Part argument, which will cause the relevant Event to return its requested node. If no Part is given as an argument, the node of an indeterminate Event in the Moment will be returned. In this case, a warning will also be displayed indicating that the node may have incorrect information. The available methods for directly accessing these nodes are:

```
Moment.measure(Part)
Moment.attributes(Part)
```

Here is an example script constructing a search for all instances of two parts with notes that start at the same moment, and the notes that precede them in the part.

```
part1 = SQLsetup(**options) # <-- This is in the
  template
part2 = Part()
moment = Moment()
note1 = moment.add_note(part1)
note2 = moment.add_note(part2)
p_note1 = note1.add_previous_note()
p_note2 = note2.add_previous_note()
```

5.3 Adding property nodes

Each object has its own properties. Objects may also have additional property nodes attached. To list the properties and property nodes of an object type, use the "`musicsql list`" command. For instance, to see what properties are available for Notes, type "`musicsql list notes`".

Musical notation information is partitioned into nodes according to the conventions of MusicXML. A list of the standard nodes and the objects they connect to is given below.

- Part:
  - files
- Moment:
  - files
- Notehead:
  - beams
- Event:
  - files
  - measures
  - attributes
  - barlines
  - directions

Other types of musical notations, such as slurs and wedges, are accessed via two objects at once. These require special methods that use the second object to specify the position of the node:

- Moment:
  - event(Part)
- Part:
  - event(Moment)
- Notehead:
  - start_slur(Notehead)
- Event:
  - start_wedge(Event)

MusicSQL comes with a number of modules that can add calculated property nodes to objects (e.g., doubled parts, chord quality). You can list the available modules using the "`musicsql list modules`" command. To add one of these modules to your database, use the command "`musicsql add moduleName`". Depending on the complexity of the information and the size of the database, it may take some time to calculate the new node.

Advanced note: MusicSQL will automatically select between natural and outer joins when attaching nodes. If you want to force a node to be attached with a natural join, add the named argument "`join='natural'`" to the `node()` method. This will limit the results to those matches that have a non-Null value in the node table.

Please note that importing a new file into your database will automatically delete calculated nodes, and you will need to generate them again. [NOT YET IMPLEMENTED]

5.4 Adding relationships and constraints

All objects have an "`add_constraint()`" method that can be used to specify conditions that you want to set for your query. These conditions can be specific to a single object (e.g., the Moment's beat equals 1), or between objects (e.g., Note 1 and Note 2 have the same duration).

To specify a condition, you must first get a property from an object or one of its property nodes. To get an object property, use the object's "`c`" feature (i.e., "column"), followed by the property you want to access.

To get one of its nodes, use the object's "`node()`" method, and hand it the name of the node you want. Then use that node's own "`c`" feature to access its properties. Here are some examples derived from a Part (part1) and a Note (note1):

```
duration = note1.c.duration
file = part1.node('files')
title = file.c.work_title
```

(To better understand the default relationships between basic objects and their nodes, see the database design diagram in the appendix.)

There are a few things you should be aware of about Python:

- You must put single or double quotes around characters to tell Python it should treat them as a string of characters rather than the name of a variable or method in your script.
- "Equals" is written as "==" (two equal signs with no space in between). A single "=" means "is set to", and is used to put values into variables. Substituting one for the other will usually cause an error when you try to run your query!

Constraints and relationships can be built out of properties using normal mathematical expressions (e.g., ==, <, >, +, -, /, *, %). For instance:

```
moment1.c.beat == 1
note1.c.duration == note2.c.duration
note1.c.semit - note2.c.semit > 3
```

Having created an expression, you must then attach it to an object that is in the expression. For instance:

```
cnstrnt = note1.c.semit - note2.c.semit > 3
note1.add_constraint(cnstrnt)
moment1.add_constraint(moment1.c.beat == 1)
```

Advanced note: All the usual SQL expression functions can be used in the construction of constraints and relationships. You can obtain an SQL function using the "`sqlfunction`" method. For instance, "`f_abs = sqlfunction('abs')`" can then be used like "`f_abs(note1.c.semit) == 5`". For a detailed introduction to constructing expressions, see the SQLAlchemy documentation under "Constructing SQL Queries via Python Expressions" (`http://www.sqlalchemy.org/docs/sqlconstruction.html`). For advanced query building you should import musicsql instead using "`import musicsql.alchemy as alchemy`". Then all musicSQL and SQLAlchemy commands can be accessed via the "alchemy" package.

5.5 Selecting properties to return

So far all of this work appears to be wasted effort, because after running the query nothing is returned! In order to get data out of the query, you must specify which properties should be output. There are two methods that can be used on any object or node, and one that is used directly from the alchemy library.

To export a property in an object or one of its nodes, use its "`add_columns()`" or "`add_column_alias()`" method. To export properties exactly as they are named in the object, hand a list of properties to "`add_columns()`". To export a property with an alias you choose, first give the name of the property to "`add_column_alias()`", then the name you want to substitute. This is especially important if you want to export the same property from two objects of the same kind! For instance:

```
part1.add_columns('part_name')
part1.node('files').add_columns('path', 'work_title',
  'movement_number')
note1.add_column_alias('semit', 'semit1')
note2.add_column_alias('semit', 'semit2')
```

You might also like to return a value that is calculated from one or more properties. Since these values are not already part of an object or node, you must use the "`add_detached_column()`" method to add it to your query. To do this, first provide the calculation (or the variable you saved the calculation in), and then the name you want that column to have. For instance:

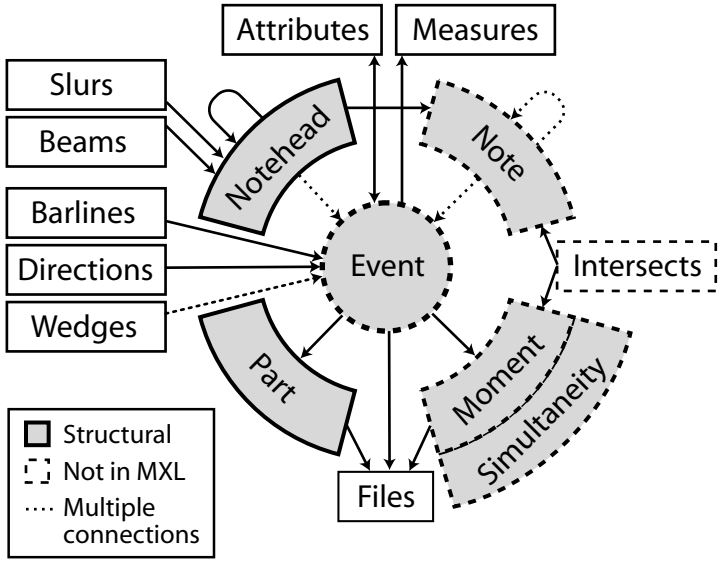
```
add_detached_column(note1.c.semit - p_note1.c.semit,
  'melint')
harmint = note1.c.semit - note2.c.semit
add_detached_column(harmint, 'harmint')
```

And that's pretty much it! At this point you should be able to start constructing your own queries and searching for matches in databases. See what you can do!

Later on you might start writing modules that add new properties to your databases. If you do, be sure to share them around.

> Advanced note: For a Python programmer familiar with classes, it isn't complicated to convert a query script into a node module. Examples and templates can be found in the musicsql.database package. You will also find templates for custom function and aggregate function modules.

6 Appendix: the basic database design



Simultaneities are only built into Tiers II and III, and do not exist in Tier I. In Tier II Simultaneities are subclasses of Moments.

[COMING SOON: A complete list of nodes and their properties.]

7 Roadmap

Planned changes or problems to fix in future versions of the MusicSQL ORM:

- Improve speed of MusicXML importing by inserting changes to database measure-by-measure instead of row-by-row.

- Add command to musicsql to delete data from a database.
- Implement triggers to automatically delete property nodes when importing new data.
- Add more property modules.

If you have any bug reports or suggestions, please email the author:

Bret Aarden <aarden@music.umass.edu>