MusicSQL ReadMe File

Bret Aarden

August 12, 2009

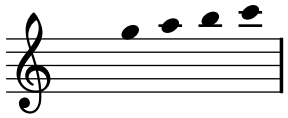
Contents

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

1 A brief introduction

MusicSQL is a system for conducting complex searches of symbolic music databases. It is designed to be flexible, extensible, and to run on industry-standard technologies.

At its simplest, the user can query MusicSQL for passages such as:



This example will be explained in more detail later on, but it is enough to know that the noteheads represent note pitches with no particular durations. After running this search on a database of Mozart string quartets, we might get a result that looks like this:

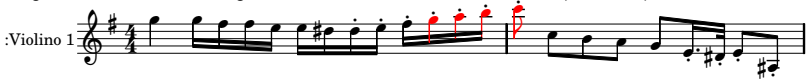
_file	_measures	_parts	_noteheads
k156-02.xml	21-22	P1	1935,1936,1937,1938
k157-01.xml	13-13	P1	3011,3012,3013,3015
k157-01.xml	14-14	P1	3021,3022,3023,3025
k157-01.xml	46-47	P1	3234,3235,3236,3237
k157-01.xml	87-87	P1	3495,3496,3497,3499
k157-01.xml	88-88	P1	3505,3506,3507,3509
k387-01.xml	133-133	P1	16702,16703,16704,16705
k387-02.xml	17-18	P1	20117,20118,20119,20120
k387-03.xml	13-13	P1	21717,21718,21719,21720
k387-03.xml	18-18	P1	21756,21757,21758,21759
k387-03.xml	21-21	P1	21790,21791,21792,21793

```
k387-03.xml   44-45        P1          22006,22008,22009,22010
k387-03.xml   75-75        P1          22173,22174,22175,22176
k387-03.xml   101-102      P1          22455,22456,22457,22458
k387-03.xml   104-105      P1          22482,22483,22484,22485
k387-04.xml   242-242      P1          25063,25064,25065,25066
```

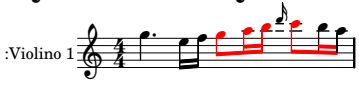
In the output, each line represents a location where this melodic fragment was found. The columns tell us which file the fragment was found in, which measures, and which part. It also includes internal information about which noteheads match the pattern: this is included in case you want to make previews.

If you also asked for previews, each output line would generate a graphic file like these:

Quartet K. 156: Quartet No. 3 for 2 violins, viola, and violoncello



Quartet K. 157: Quartet No. 4 for 2 violins, viola, and violoncello



It is possible to create much more complicated queries involving polyphony and multi-dimensional searches, but we'll get to those later.

Under the hood, MusicSQL is built on a database that contains music imported from MusicXML files. There are two ways to interact with this database. First, we can use the `musicsqlcmd.py` script on the command line for a number of basic commands. Second, we can write custom queries using the Python scripting language. There are also plans to implement a graphical interface.

How should we pronounce MusicSQL? SQL stands for 'structured query language'. In the 70s it stood for something else and it made sense to pronounce it 'sequel'. Traditional database programmers often say 'ess-kue-el', although Windows types call it 'sequel'. Let's stay mainstream and pronounce it 'music ess-kue-el'.

2 Installing MusicSQL

We will probably need to install a few pieces of software on your computer before using MusicSQL:

- The database server
- The Python interpreter (Windows only)
- The LilyPond music typesetter (optional)
- MusicSQL and additional Python packages

In the future it may be possible for an administrator to set up MusicSQL on a single server, and allow multiple clients to submit queries remotely. For now, it is simplest to install all the software on a single machine.

2.1 The database server

By default, MusicSQL assumes that you have the MySQL database server running on your computer. The MySQL Community Edition is a professional open-source database server that is available for free and runs as a background process. It is usually easy to install, either via an installer package (for Mac or Windows) or a package manager (for Linux). You can find installation information on the MySQL website:

`http://dev.mysql.com/downloads/mysql/5.1.html#downloads`

Mac: Follow the installer directions to install the server. Then double-click on the MySQL.prefPane that came with 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.

Windows: Run the main installer, 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.

Linux: On Ubuntu the package is called 'mysql-server'.

You will probably want to set a password on the server if your computer can be accessed by other machines. In that case you will need to provide the password every time you issue a MusicSQL command. If you are comfortable with your computer's network security it is possible to leave the password blank.

You can also use MusicSQL with other database servers, such as PostgreSQL or SQLite. PostgreSQL has comparable performance to MySQL, but is somewhat harder to install. SQLite can be easy to install, but it has limited ability to optimize queries, which makes it unusable for many tasks. To select one of these other servers, use MusicSQL's '`--backend`' option followed by '`postgres`' or '`sqlite`'. Other databases such as Oracle, SQL Server, and Firebird may be supported in the future.

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

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

2.2 The Python interpreter

MusicSQL is written in Python, a popular interpreted programming language. A Python interpreter comes standard on most operating systems. MusicSQL requires at least Python version 2.5. You can check which version you have installed by typing '`python`' on the command line. Version 2.5 started shipping with Mac OS 10.5 (Leopard) and Ubuntu 8.04 (Hardy Heron).

Users of non-Microsoft operating systems can skip to the next subsection. (Or you may read on in macabre fascination.)

Microsoft did not write its own version of Python, so it is not included with Windows and must be installed manually. A Windows installer can be downloaded from the Python website:

`http://www.python.org/download/`

Python will not always work out of the box in Windows. If you type '`python`' at the Command Prompt and get an error like '`'python' is not recognized`', then you need to edit your environmental variables.

Windows XP: Click on the Start menu and choose the Control Panel. In Classic view, choose System; in Category view, choose Performance and Maintenance, then System. Click the Advanced tab, then Environmental Variables. Choose 'Path' from the System Variables, then click Edit.

Add the Python directory (usually '`C:\Python25`' or something similar) to your PATH variable. If you installed LilyPond first then it probably automatically put its own embedded Python interpreter onto PATH, but that version of Python lacks the ability to install new libraries, and you must put the Python you just installed ahead of LilyPond on the PATH.

When we get to installing MusicSQL, you may find that when you try to run `musicsqlcmd.py` you get an error such as '`ImportError: No module named musicsql`', or Windows may ask you what program to open it with. You should be able to fix these problems by editing a registry value.

Windows XP: Click the Start menu and choose 'Run...'. Type '`regedit`' and click OK. In the left column, navigate to `HKEY_CLASSES_ROOT\Python\shell\open\command`. Double-click the item in the right column and edit the path so it points to the Python you just installed.

2.3 The LilyPond music typesetter

Although not strictly necessary, if you want to be able to automatically produce music notation from a query you must have the LilyPond music typesetting software installed. LilyPond installers are available from the website (or the Linux package manager):

`http://lilypond.org/web/install/`

MusicSQL assumes that LilyPond is installed in the default location for your OS:

Mac:	/Applications/
Windows:	C:\Program Files
Linux:	/usr/bin

If you choose to put it somewhere else (or if your package manager puts it in a creative location), you will need to edit the CONFIG file in the MusicSQL folder.

2.4 MusicSQL and additional Python packages

All software uses libraries written by other people. Like UNIX, Python is designed to distribute those libraries separately from the programs that use them. Python does not currently have a built-in system for automatically installing libraries, but the manual process isn't usually too hard. We will need to install the following library packages:

- SQLAlchemy
- Your database driver
- MusicSQL

The following steps require the use of a command line. The details of how to access it differ by operating system. *Mac:* The Terminal.app is the default command-line program, and it's located in the /Applications/Utilities folder. *Windows:* The Command Prompt application can be run by clicking on the Start menu, then All Programs, then Accessories, then Command Prompt. *Linux:* All Linux systems are different. On Ubuntu, click the Applications menu, then Accessories, then Terminal.

2.4.1 SQLAlchemy

SQLAlchemy is the premiere SQL database interface, arguably for any programming language. When we start building queries from scratch a lot of things will Just Work thanks to this library.

Mac/Windows: You can download SQLAlchemy from its website:

```
http://www.sqlalchemy.org/download.html
```

Then unarchive the file. *Mac:* Double-click on the file. *Windows:* You will need to download a unarchiving utility like 7-Zip to unarchive it (`http://www.7-zip.com`).

On the command line, navigate to the new sqlalchemy folder. First determine what folder your command line is currently in. (*Mac:* Type '`pwd`'. *Windows:* The prompt should tell you what the current folder is.) Type '`cd ..`' to navigate up. To navigate down, type '`cd foldername`', where 'foldername' is the folder you want to navigate into. Often if you type the first few letters of the folder and hit Tab, the rest of the name will be completed automatically. Continue until you arrive in the sqlalchemy folder.

Install the package by typing '`python setup.py install`'. *Mac/Linux:* If you get a permissions error, put the word '`sudo`' first. You will be asked for the administrator's password. On a Mac this is commonly your own password.

Linux: you may be able to install SQLAlchemy using your package manager. In the case of Ubuntu, the package is named 'python-sqlalchemy', and it is not part of the standard repositories. You'll need to activate the 'universe' repository first. If the package manager can't do it, follow the above directions.

Let's test whether the installation worked. Start your Python interpreter by typing '`python`' on the command line. At the '`>>>`' prompt type '`import sqlalchemy`'. If you don't get an error, you're set.

2.4.2 Your database driver

Each database has a different Python driver, and sometime more than one. Here are database drivers that are compatible with SQLAlchemy:

MySQL:	`MySQLdb`
PostgreSQL:	`psycopg2`
SQLite:	`sqlite3` (included with Python 2.5+)

Database drivers need to be fast, so they are usually custom built for each operating system.

Windows: Here's where it's nice to have Windows – other people have compiled versions of the drivers for you. You can get them at these locations:

MySQL:	`http://sourceforge.net/projects/mysql-python/` (use the default download link)
PostgreSQL:	`http://www.stickpeople.com/projects/python/win-psycopg/`

Mac: If you have an Intel machine running OS 10.5 (Leopard), you can download an installer for the MySQL driver here:

`http://faculty.ncf.edu/baarden/musicsql/MySQL_python-1.2.3c1-py2.5-macosx-10.5-i386.egg`

To run the installer, navigate to its folder and type '`easy_install filename`' where 'filename' is the file you just downloaded. If you get a permissions error, put the word '`sudo`' first. You will be asked for the administrator's password. This is commonly your own password.

Sadly, there are currently no other pre-built installers for Mac Python drivers, so to install a driver for another database or processor you need to have Xcode installed first. It's an optional 1GB install on the OS installation DVD. (If you're registered on Apple's Developer Connection, you can also download a copy from their website.) To install a driver manually, download it from the Linux link below, unarchive it, '`cd`' to that folder, and type in the '`python setup.py install`' command. Again, if you get a permissions error put the word '`sudo`' first. Given the current state of the drivers, the install may fail. If so, a web search should reveal a fix. With Xcode installed you could use the MacPorts or Fink package manager to install the driver, but then it would be linked to the package manager's version of Python, which the `musicsqlcmd.py` file will not use unless you manually edit it.

Linux: First check whether your driver is already installed. Use the same technique as for sqlalchemy, above: start the Python interpreter, then type '`import MySQLdb`', or whatever driver you're checking for. If it's not there, try to install it using your package manager. If you can't, then download the driver from the link below, unarchive it, '`cd`' to that folder, and type in the '`python setup.py install`' command. If you get a permissions error, put the word '`sudo`' first. You will be asked for the administrator's password.

MySQL: `http://sourceforge.net/projects/mysql-python/` (Click on 'View all files' and select the version ending in '.tar.gz')

PostgreSQL: `http://initd.org/pub/software/psycopg/` (Select the highest version number)

2.4.3 MusicSQL

Presumably you already have the MusicSQL installer downloaded and unarchived. To install, simply navigate to its folder in the command line and type the '`python setup.py install`' command. *Mac/Linux:* If you get a permissions error, put the word '`sudo`' first. You will be asked for the administrator's password. On a Mac this is commonly your own password.

Let's test whether MusicSQL installed correctly. At the command line, type '`musicsqlcmd.py`'. You should get a response that looks like this:

```
usage: musicsqlcmd.py (setup|import|ezsearch|export|list|
    add|delete|preview|previewxml|kill|template) ...
```

3 Getting started

Now that the software is installed, let's walk through a first MusicSQL query. We'll deal with three steps:

- Setting up a database
- Importing data
- Your first query

All the instructions in this section assume that you're typing into the command line. If you set up your database with a password, you will need to add the '`--password`' argument to the end of each line, and `musicsqlcmd.py` will prompt you for the password.

3.1 Setting up a database

Database servers hold more than one database. If we want, we can create a separate database for each kind of music, or insert all our music into one database. For our first database, we'll use the Schubert string quartets available in MusicXML form on the MusicSQL website, and we'll put them in a database called '`schubert_quartets`'. To begin, type this command:

```
musicsqlcmd.py setup --database schubert_quartets
```

The last pair of words in the command is known as a "named argument", and consists of a name preceded by two hyphens, followed by the value we are assigning to that name. Often you can shorten the name to a single hyphen and the first letter of the name (e.g., `-d schubert_quartets`), but only if the single letter is unambiguous.

If all goes well (that is, if you don't get an error), we now have a new, empty MusicSQL database on the server.

3.2 Importing data

There are many music notation file formats, but the emerging standard is MusicXML. 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 has been tested with MusicSQL.

`http://www.gutenberg.org/browse/categories/4`
`http://www.cpdl.org`
`http://extras.humdrum.net/man/hum2xml`

For now, let's work with the Schubert string quartets that are available for download on the MusicSQL website. Once you have downloaded and unarchived them, navigate to that folder in the command line. We'll start by importing the first movement of Schubert's String Quartet no. 1. Type the following:

```
musicsqlcmd.py import D18_1.xml --database
    schubert_quartets
```

You should see the program report the status of the import, and finally end with a comment such as '`Completed in 15 seconds.`' You may also see some warnings (usually because the program that exported the MusicXML mistook a mid-measure barline for the end of a measure), but those should not be a problem for now as long as the import completes successfully.

To drop this database from the server, type the following:

```
musicsqlcmd.py delete database --database
    schubert_quartets
```

If you were to re-create the database, you could then import all of the quartets in the folder at once with this command, where the '`*`' is a wildcard that matches anything:

```
musicsqlcmd.py import *.xml --database schubert_quartets
```

3.3 Your first query

To begin searching, let's start with the basic query shown in the introduction. We can run that search on our new database with the following basic command:

```
musicsqlcmd.py ezsearch 'G5 A5 B5 C6' --database
    schubert_quartets
```

The results should be analogous to those in the introduction. To break it down piece by piece:

`musicsqlcmd.py`	:	this starts the program.
`ezsearch`	:	this gives the command to run a basic search.
`'G5 A5 B5 D6'`	:	the successive notes we are searching for, enclosed in quotes.
`--database schubert_quartets`	:	we tell the program which database to search.

If we want to see notation previews of the results, we just add the `--preview` option to the end of the command.

3.3.1 Pitches

If you are familiar with the so-called 'scientific pitch notation', then you recognized the pitch format used in our searches thus far. The complete pitch specification is, in order from left to right:

Step	a letter (uppercase or lowercase) indicating the scale step
Alter	zero or more octothorpes ('`#`') or hyphens ('`-`') for sharps or flats, respectively
Octave	an optional number indicating the octave in scientific pitch notation

As with all Western pitch systems, each new octave starts on C, so '`A#3 B3 C4`' is a stepwise upward pattern. The same pattern in reverse might read '`C4 B3 B-3`'.

3.3.2 Durations

If both a duration and pitch are included, the duration is placed before the pitch. Durations are notated using the Humdrum 'recip' (reciprocal) representation. The complete duration specification is, in order from left to right:

Reciprocal value	the number of these durations that would fit in a whole note
Dots	zero or more periods (‘`.`’) for standard rhythmic dot notation

A list of common reciprocal values is given below. Note that tuplet values are handled implicitly: the value simply indicates how many would fit in a whole note.

1	whole note
2	half note
3	triplet half note
4	quarter note
5	quintuplet half note
6	triplet quarter note
8	eighth note
10	quintuplet quarter note
12	triplet eighth note
16	sixteenth note
20	quintuplet eighth note
24	triplet sixteenth note
32	thirty-second note

In the ‘`ezsearch`’ method, durations are treated without regard for whether a note is broken up into tied noteheads. At present, reciprocal numbers below the whole note level are not yet supported.

A more complete specification of the recip notation system is available here:

`http://music-cog.ohio-state.edu/Humdrum/representations/recip.rep.html`

So far we’ve simply typed in some commands at random. In the next section we’ll see the full capabilities of the `musicsqlcmd.py` program.

4 An overview of musicsqlcmd.py

The ‘`musicsqlcmd.py`’ script provides a basic interface to the MusicSQL libraries and database backends. To get usage information about `musicsqlcmd.py`, simply type ‘`musicsqlcmd.py`’ at the command line. This will show you a list of available commands. Each `musicsqlcmd.py` command has its own usage information that can be accessed by typing ‘`musicsqlcmd.py command`’. (Replace ‘command’ with the name of the command you are interested in.)

Here is a quick overview of the available commands:

`setup`	create a new database on the server
`import`	import a MusicXML file into the database
`ezsearch`	run a basic search of pitches and/or durations
`export ...`	export part (or all) of a file in the database in MusicXML format
`list databases`	list all MusicSQL databases on the server
`list files ...`	list all MusicXML files imported into the database
`list nodes ...`	list all the information nodes in the database
`list properties ...`	list all the information properties in a node
`list values ...`	list all the distinct values of a property
`list modules`	list all the optional information nodes that can be added to a database
`add ...`	add an optional information node to a database
`delete database ...`	delete a database and all its information from the server
`delete file ...`	delete all the data from one file from a database
`preview file`	create previews from query results
`kill pid`	stop the query with the thread ID 'pid'
`template`	print out a query script template

We've already seem a couple of named arguments that can be used with `musicsqlxmd.py`. Below is a list of the standard arguments. Some of these are optional because they have useful default values, shown here in square brackets. Keep in mind that some commands have additional named arguments available; you can learn about these by accessing the usage information on the command line.

`--database name`	the database to connect to
`--host address`	where the database server is located [`localhost`]
`--user username`	the server login name [*server specific*]
`--password`	ask for the server login password
`--backend name`	what kind of database server you use [`mysql`]
`--previewdata`	add columns to the output required to export previews
`--preview`	automatically create previews along with the results
`--format extension`	specify which LilyPond graphic format to use [`png`]
`--verbose`	print technical information

5 Advanced queries

Although it is possible to search a MusicSQL database simply using the '`ezsearch`' command of `musicsqlcmd.py`, the real power of MusicSQL is its ability to build arbitrarily complex searches using a fairly simple vocabulary. The examples in this

section will lead you through several layers of complexity:

- Our first query script
- Our first polyphonic query
- Our first multi-dimensional query

The examples will continue to include visual representations of the search patterns to guide the discussion. Eventually an analogous visual search client may be implemented, but for now these graphics will serve to help us visualize the MusicSQL objects.

5.1 Our first query script

The scripting approach to MusicSQL queries requires that we edit text files. To begin, we will ask `musicsqlcmd.py` to provide us with an empty script template. Type this command to create the script:

```
musicsqlcmd.py template > query1.py
```

Let's break this down:

`template`	A command to print out a basic query template
`> query1.py`	Tells your computer to dump that text into a file named '`query1.py`'.
`.py`	This file extension informs us and the computer that this file is a Python script.

The next step is to open the '`query1.py`' text file in a text editor. You can use a basic text editor that comes with your operating system (*TextEdit* on a Mac, *Notepad* on Windows, *Text Editor* in Ubuntu Linux), but there are free specialized editors that make writing scripts easier, like *Komodo Edit*, which is available for Mac, Windows, and Linux:

`http://www.activestate.com/komodo_edit/`

An editor like *Komodo Edit* can read the MusicSQL library and prompt you with the names of objects and methods you can use while writing your script. You can also run the script within the editor so that you can avoid the command line if you prefer.

Mac: The current version of *Komodo Edit* does not read user-installed libraries such as MusicSQL correctly. To fix this, click on the Komodo menu, then Preferences. Open the Languages arrow in the left column, and click on Python. Under 'Additional Python Import Directories', click the plus button. Navigate to the top of your hard drive, click on Library, Python, 2.5 (or the newest version, if it's higher), and site-packages, then click Open. Click OK to close out of Preferences, and shortly it will start dynamically suggesting completions to your code.

The contents of '`query1.py`' should look like this:

```
#!/usr/bin/python

import musicsql
q = musicsql.Query()

### CONSTRUCT YOUR QUERY HERE ###
part1 = q.part()
note1 = part1.add_first_note()

### END OF QUERY ###

q.print_run()
```

The only part of this file that we're concerned with is the bit after '`CONSTRUCT YOUR QUERY HERE`'. There are a couple of lines there already which begin with '`part1 = q.part()`' — this is the starting point of all queries. Here is how to understand what they mean:

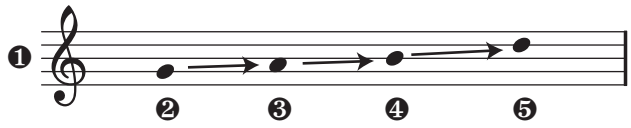
`part1 =`	We create a new object named '`part1`' and define what it is.
`q`	The MusicSQL query object which holds the query we construct.
`.part()`	A method built into the Query object that creates a new Part.
`.add_first_note()`	A Part method that creates the first note.

Much of the process of building a query involves creating a new object and defining it using a method from another object. To call an object's method we use 'dot-notation', which means putting a dot between the object name and the method. Methods are followed by parentheses, and you can send information to a method by putting it between the parentheses. Always remember to put parentheses after methods.

When working with objects, it is helpful to differentiate the abstract object from a particular object that represents something in our query. One convention is to capitalize objects like Parts and Notes to indicate they represent the class of object, and to use lowercase — like `part1` and `note2` — to indicate a particular instance of that object. (In database terms, we can imagine the Object represents a table, and an `object` is a row in that table.)

Part objects have a special method called '`add_first_note()`' that is only called once at the start of the query. It creates a Note out of nothing. Every other element in the query must be constructed relative to that first Note in some way. The only exception to this are Parts: any number of them can be constructed directly from the Query object.

Suppose we wanted to search for the ascending pitch sequence pictured below: G4, A4, B4, D5. The numbered steps (as labeled in the figure) 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 include them.



1. Every MusicSQL query starts with a Part which we get from our Query object using its '`add_part()`' method. In the diagram the Part is pictured as a staff. Even though it is notated 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, where '`q`' is our Query object.

```
part1 = q.part()
```

2. The template gives us a Note which we get from a Part using its '`add_first_note()`' method. We can add an argument to the method to set the pitch to '`G4`'. If we pass a string with note information to this method, it will require the note to match that information. 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_first_note('G4')
```

3. We then ask the Note to give us its next Note on A4.

```
note2 = note1.add_next_note('A4')
```

4. We then ask that Note to give us its next Note on B4.

```
note3 = note2.add_next_note('B4')
```

5. Finally we ask that Note to give us its next Note on D5.

```
note4 = note3.add_next_note('D5')
```

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. Here is another way of writing the same query:

```
part1 = q.part()
note1 = part1.add_first_note('G4')
note2, note3, note4 = note1.add_note_sequence('A4',
   'B4', 'D5')
```

5.1.1 Running the query script

To run this query, we must now execute the script. Although there are shortcuts, the failsafe way to do it from the command line is this:

```
python query1.py --database schubert_quartets
```

Notice that just like the `ezsearch` command, we need to include the `--database` argument to tell our script which database to search. Typing this in will cause MusicSQL to go through the motions, search the database and then — return nothing.

It is up to us to select which information to query. There is a lot of information stored in the database, and not all of it is useful for a given query. We've already seen one shortcut that produces output, and that's the '`--previewdata`' option. Add that to the command line, and we should see something like this:

```
Executing query (13 joins, thread ID 120)...
Fetching results...
Progress: 45%
Progress: 46%
2 results.
_file     _measures _parts    _noteheads
D46_1.xml 72-72     P1        17305,17306,17307,17308
D46_1.xml 58-58     P2        18530,18531,18532,18533
```

Komodo Edit users can execute the script within the editor. Click on the Tools menu and choose 'Run Command...', which will pop up a dialog with a 'Run' textbox where you can type in a command. To the right of the textbox is an arrow that will pop up a Shortcuts menu. From the menu choose 'Configured Python Interpreter' ('`%(python)`'), and add a space after it in the textbox. Choose 'file path' ('`%F`') from the menu, and add another space. After that you can add arguments like '`--database`' and '`--previewdata`' to the textbox. When you click the Run button a Command Output tab should pop up to show you the results. Be aware that if you haven't saved your file this will run the last version you saved, not what appears in the editor. Unsaved files show an asterisk in the tab.

5.1.2 Selecting additional information

Objects have several methods to select information for output, which are shown in the table below. The names of the properties being selected should be passed to these methods as strings (enclosed in quotes).

`.select('property1', 'property2', ...)`	Add one or more of the object's properties to the output.
`.select_alias('property', 'alias')`	Add one of the object's properties to the output, but use an alias for its header.
`.select_all()`	Add all of the object's properties to the output.

Recall that we can find list the properties in a database node by using the '`musicsqlcmd.py list properties`' command. Each object has a corresponding node in the database, and all nodes of the same type share one table. We have encountered parts and notes so far, and their nodes in the database have the names '`parts`' and '`notes`', respectively.

The '`select_alias`' method is especially useful when there are multiple objects of the same type, since if we select the same property name from multiple objects only one of them will be output. By using an alias for one or both, the same property can be output from multiple objects. The '`select_all`' method is a convenient way to output all the properties of an object, but none of the properties selected this way will be aliased.

5.2 Our first polyphonic query

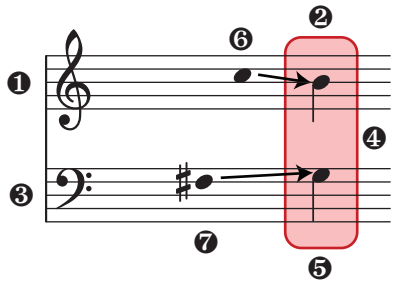
We can begin to see the flexibility of a MusicSQL search as we move on to polyphonic searches. When searching for melodies it was possible to think of object relationships

sequentially, as with the '`add_note`' method. In order to construct vertical relationships, MusicSQL has Moment objects that represent a slice of time. Note objects have a '`start_moment()`' method that will return the Moment that occurs at the onset of the Note.

We also want to start thinking about how to use a query constructed in MusicSQL. One way to approach a query is to treat it as a research question. The first step is to *define our terms,* which means setting up the mechanics of a query, like we did in our first query script. By precisely defining how the search is constructed, we better understand what we are searching for. The second step is to *frame our question,* or decide what information about the search results we want to select from the database. Let's look at each of these in turn.

5.2.1 Defining our terms

Suppose we wanted to search for a specific cadential arrival in G major in two voices, as shown in the figure below. The colored box in the figure represents a Moment spanning all the Parts, indicating that the two Notes begin at the same Moment. We'll require that both of these Notes are quarter notes, but we won't specify the durations of the two prior notes approaching in contrary motion. We will also leave the onset times of the two prior notes flexible.



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

```
part1 = q.part()
```

2. We'll tweak the '`add_first_note`' method to specify a particular duration and pitch.

```
note1 = part1.add_first_note('4B4')
```

3. Another Part can be generated separately; Parts are the only objects that can be generated directly from a Query object. Although the figure above shows the second Part with a bass clef, that is not actually defined in our search.

```
part2 = q.part()
```

4. We want to specify that two Notes occur at the same Moment, so we need to get the Moment of our Note's onset using its '`start_moment()`' method.

```
moment = note1.start_moment()
```

5. Now that we have the Moment, we can use it to create a Note in the other Part as well. The Moment's '`add_note`' method requires that we tell it what Part the Note goes in. We also have the option to specify that this Note should be a half note on G3.

```
note2 = moment.add_note(part2, '4G3')
```

6. We then ask the first Note to add the Note on C5 that precedes it with its '`add_previous_note`' method.

```
prevnote1 = note1.add_previous_note('C5')
```

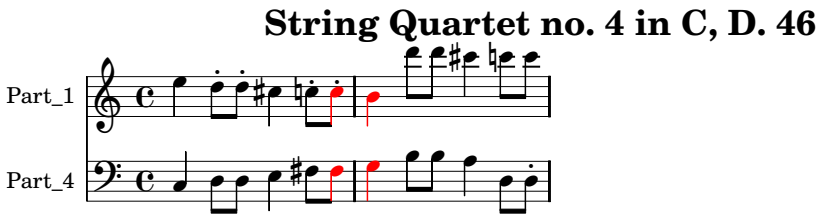
7. And then we ask the Note in the second Part to add the Note on F♯3 that precedes it.

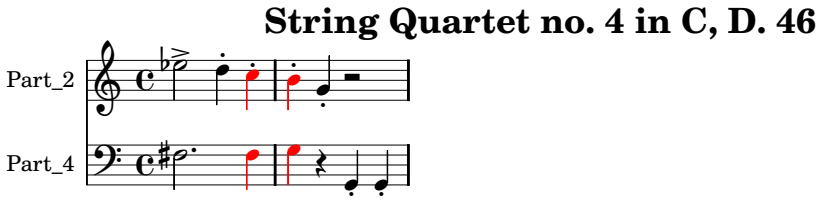
```
prevnote2 = note2.add_previous_note('F#3')
```

Running this query with the '`--previewdata`' option should result in something like the following:

```
Fetching results...
Progress: 45%
Progress: 46%
2 results.
_file     measures  _parts    _noteheads
D46_1.xml 122-123   P1,P4     17478,17479,21236,21237
D46_1.xml 161-162   P2,P4     18945,18946,21400,21401
```

Using the `--preview` option instead might product results like these:





5.2.2 Framing our question

Now that we have a way to search for this kind of cadential pattern, what is it we want to know about it? Perhaps we could ask what beat this pattern tends to occur on. In order to pose this question, we need to know a little more about how information is structured in MusicSQL.

One design principle of databases is that information should not be duplicated. The 'beat' property is not stored in Note objects, because the same information would be duplicated across several notes. Instead it is stored it a little higher up. Moments would seem to be the place to store beat information, but it is possible that each Part could have a different time signature, so a single moment could represent a different beat in each part.

What we need is an object that represents a Moment within a specific Part. In MusicSQL this intersection of a Moment and a Part is known as an Event. Notes onsets occur inside Events, and multiple Notes can share onsets in the same Event (we could call them chords). An Event is a very useful structural and informational nexus, as we shall see.

An Event by itself has only one property, but it's the one we were looking for: a '`beat`'. To obtain it we need simply access the Event and select its property. Note objects have an '`start_event()`' method to return their onset Events.

```
event = note1.event()
event.select('beat')
```

The beat value might mean different things depending on the meter, so we should also find out the meter of each result. Meter information is an 'attribute' in MusicXML, so it is stored in the '`attributes`' table. Attributes are specific to a Part and change over time, so it makes sense that the `attributes` node is connected to the Event hub. To get access to a node, we use the '`node()`' method common to all Hubs. In MusicXML the numerator of the meter is the '`beats`' property, and the numerator is called '`beat_type`'. (Technically, MusicXML uses hyphens rather than underscores, but SQL doesn't like hyphens.)

```
attributes = event.node('attributes')
attributes.select('beats', 'beat_type')
```

Let's take our search in a different direction. Perhaps we're interested in knowing whether dynamics are handled consistently at these cadence points. In MusicXML

dynamics are specific to individual notes, so we can expect to find this information below Events at the Note level. In fact, however, we need to go even one level lower to the Notehead object.

We've been comfortably oblivious of the Notehead object to this point because we haven't asked questions about how exactly the music appears on the page. We've been using Note objects instead, which are an abstraction with no analogue in MusicXML. The Note exists purely for the convenience of searching. For instance, for transposing instruments there is no place in a MusicXML file where the concert pitch of a note is encoded. All pitches in Note objects are in concert pitch, since for most purposes we are more interested in how a note sounds rather than how it appears on the page. It's also usually not important whether a 2-beat duration is notated as a half-note or two quarter-notes tied together, so a Note object collapses tied notes together and simply measures the total duration. When we set the duration of a Note to a quarter note earlier, it actually represented a quarter-note duration rather than a place in the score where a quarter note appears.

If we need information that is specific to how the music appears on the page, we can seek that out in Notehead objects. Note objects have a '`notehead`' method that returns the Notehead that shares the same onset. The dynamics might differ between notes, so let's select from both Noteheads. Because they both have the same property name, we'll need to give them aliases so that both are selected (and so we can tell them apart).

```
notehead1 = note1.notehead()
notehead2 = note2.notehead()
notehead1.select_alias('dynamics', 'dynamics1')
notehead2.select_alias('dynamics', 'dynamics2')
```

As it turns out, there is no dynamics information at those points in the Schubert quartets database, so the values in the dynamics columns should be '`None`'.

Now we have a fairly complete picture of the relationships among MusicSQL objects. To condense it, we can imagine a hierarchy of objects that represent the music from concrete to abstract and small to large.

The information about which Hubs and Nodes contain what properties has been pulled apparently from nowhere, but you can look it up yourself. We saw earlier that `musicsqlcmd.py` can provide the information, but a complete listing of nodes and properties is also given at the end of this document in Appendix A. Similarly, a complete listing of object methods is provided in Appendix B.

5.3 Our first multi-dimensional query

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.

Over the course of constructing this query, we'll cover several new topics, including:

- Adding module nodes
- Constructing expressions
- Conditional logic
- Simultaneities
- Selecting Conditionals

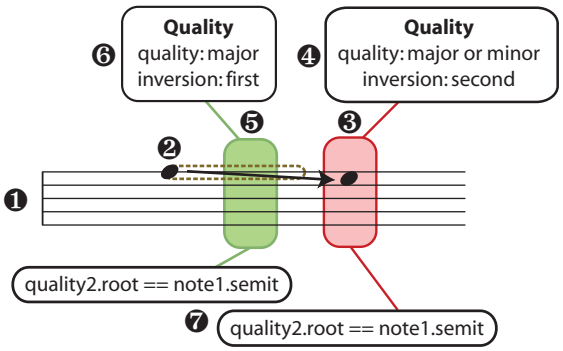
5.3.1 Adding module nodes

There is no chord quality property in the basic database, since MusicXML doesn't have one. But one of the features of the MusicSQL system is the ability to run modules that add information nodes to the database. One module that comes with the MusicSQL distribution is the 'quality' module, which calculates the quality, root, and inversion of each Moment in the database. Once a node is added to the database, we can access it through its Hub module(s) — in this case, the Moment hub. The convenient way to add a node is via the `musicsqlcmd.py` tool:

```
musicsqlcmd.py add quality --database schubert_quartets
```

The list of modules that come standard with the MusicSQL distribution is shown at the end of Appendix A, and is also available through the '`musicsqlcmd.py list modules`' command. Constructing your own module is fairly easy (but not trivial): any query can be turned into a module that adds its results to a table in the database. New modules can be distributed and even incorporated into the standard MusicSQL distribution..

Let's continue with our multi-dimensional query. 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, where the two notes represent the Neapolitan root resolving to the tonic. What follows is one possible procedure for constructing the pattern in MusicSQL.



1. As usual, we start by getting the first Part.

```
part1 = q.part()
```

2. Then we add a Note to the Part.

```
note 1= part1.add_first_note()
```

3. To construct the characteristic voice-leading, we add the following Note and specify that it moves down by one chromatic step. This can be done using the Note's 'set_chromatic_interval' method.

```
note2 = note1.add_next_note()
note1.set_chromatic_interval(note2, -1)
```

4. Having added the quality node to the database, we can now access that information to set constraints on our search. Let's start by specifying that at note2's onset the Moment should contain a second-inversion major or minor chord.

First we'll get the Moment using the Note's start_moment() method. From there we can use the Moment's node() method to get the quality node.

```
moment2 = note2.start_moment()
quality2 = moment2.node('quality')
```

5.3.2 Constructing expressions

To refer to a particular property of a node we use its property() method. For a second-inversion chord, the 'inversion' property of the quality node is 'second'. (If you're unsure what all the values of a property are, you can use the 'musicsqlcmd.py list properties' command to check.)

Given this information, we can now construct our expression. In a MusicSQL script, basic expressions are written just like mathematical formulas:

```
quality2.property('inversion')== 'second'
```

Note that in Python the 'equals' symbol is written as '==' to differentiate it from the act of assigning a value ('=').

An expression can be used for many things, but by itself it does nothing. If we want to use it to constrain our query, we need to use a Hub object's '`add_constraint()`' method. Keep in mind that it only makes sense to constrain an object using an expression that it is related to. If we were to try to add this constraint to a Note object, for instance, MusicSQL would complain.

Here is the actual command that constrains the Moment to second inversion:

```
moment2.add_constraint(quality2.property('inversion') ==
    'second')
```

5.3.3 Conditional logic

We also want to constrain the Moment to either major *or* minor chords. To construct more complicated relationships that involve if/then or and/or logic, MusicSQL provides a special Conditional object. These objects can be generated from a Query object in a way that's analogous to how we create a Part: we simply call its '`conditional()`' method. Conditionals act just like other expressions, and you can even use a Conditional as an expression for another Conditional.

Here is a brief listing of the methods of the Conditional object:

Method	Description
`IF(expression [,THEN=value])`	Adds an expression to the Conditional.
`THEN(value)`	The value to return if the latest IF/ELSEIF expression is True. Required if the Conditional will be used as a constraint, unless a THEN argument was already included with the IF method.
`ELSEIF(expression [,THEN=value])`	Adds another expression with its own THEN to the Conditional.
`ELSE(value)`	The value to return if none of the IF expressions are true
`AND(expression)`	Adds another expression to the latest IF/ELSEIF expression. The IF will only be True if this AND the previous expression are True.
`OR(expression)`	Like AND, but the IF will be True if this OR the previous expression is True.

The uppercase names are important so that Python won't mistake them for its own if/then/else/and/or words. These Conditional methods can be strung along end-to-end if it is convenient.

Here is how we can use a Conditional to express the OR logic in our chord quality constraint:

```
majorminor = q.conditional()
majorminor.IF(quality2.property('quality') == 'major')
majorminor.OR(quality2.property('quality') ==
    'minor').THEN(1).ELSE(0)
moment2.add_constraint(majorminor)
```

Although there are many ways to represent True and False in a database, the most universal are 1 and 0, respectively, and these are the values used for the THEN and ELSE methods above.

5.3.4 Simultaneities

5. We want to specify what harmony leads into Note 2. But in real music we can't assume that's the harmony that occurs right as 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. (In the search diagram above, the extent of the Note is represented by a dotted line, and the Simultaneity is a Moment box floating above it.)

A Simultaneity attached to one Note will match any Moment that occurs during the length of the Note. If multiple Notes are attached 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.

To add a Simultaneity to a Note we use its `add_simultaneity()` method.

```
moment1 = note1.add_simultaneity()
```

6. Then we constrain the Simultaneity (the Neapolitan) to be a major chord in first inversion.

```
quality1 = moment1.node('quality')
moment1.add_constraint(quality1.property('quality') ==
    'major')
moment1.add_constraint(quality1.property('inversion') ==
    'first')
```

7. Finally, we constrain both Notes to be the roots of their chords. We can do this by requiring that the '`root`' property of the '`quality`' node (a PC value) and the Note's '`pc`' property are the same.

```
note1.add_constraint(note1.property('pc') ==
    quality1.property('root'))
note2.add_constraint(note2.property('pc') ==
    quality2.property('root'))
```

This completes the query structure. Running the script with the '–previewdata' option should produce something like this:

```
Executing query (10 joins, thread ID 303)...
Fetching results...
Progress: 95%
1 results.
_file     _measures _parts    _noteheads
D87_4.xml 244-244   P1        48807,48809
```

5.3.5 Selecting Conditionals

Complex expression structure is not limited to the structure of a query: it can also be used to produce more useful output. Suppose we wanted to know whether the two notes depicted in our diagram are in the top voice of the texture. This information isn't part of the core database, but it can be added via the '`topnote`' module.

The `topnote` node has no accessible properties; rather, it acts as a true/false value. (See Appendix A for more details.) After a true/false node has been attached, its '`is_null()`' method returns an expression that tests whether it has a value for the Hub:

```
topnote = note2.node('topnote')
is_topnote = topnote.is_null()
```

This expression can be used to set a constraint on `note2`, but what we really want to do is select its value regardless of whether it is True or False. The Query object has a method for selecting arbitrarily expressions called '`select_expression()`'. Because expressions are nameless, you are required to provide an alias for the selection, like this:

```
q.select_expression(is_topnote, 'topnote')
```

At this point you should have a pretty good idea of how to conduct searches using the MusicSQL system. Read on into the appendices for more details of object properties and methods. If you run into situations that aren't explained in the documentation or don't seem to be supported by the MusicSQL architecture, please help with the development of this software by contributing your feedback at the project website:

`http://code.google.com/p/musicsql/`

Appendices

A Properties of objects

MusicSQL is designed to import MusicXML files, and so property names inside the database structure are largely borrowed from and have the same meanings as names in MusicXML. Rather than duplicate those same definitions here, they can be found in the MusicXML DTDs:

`http://www.recordare.com/dtds/index.html`

The Hub objects are the main objects that are used to construct query relationships. Other nodes in the database hold information that is related to the Hub objects. Here first are the properties of the Hub object tables:

Hub table	**Properties**
parts	`part_attr, part_name, part_group`
notes	`concert_step, concert_alter, concert_octave, semit, pc, duration`
noteheads	`voice, staff, notehead_duration, actual_notes, normal_notes, rest, step, alter_, octave, grace_order, steal_time, type, dot, accidental, stem, dynamics, articulation, tie_start, tie_stop`
moments	`ticks`
simultaneities	*see moments*
events	`beat`

You'll notice that some properties have unexpected underscores, such as the `alter_` property of the noteheads table. This is because certain words have reserved meanings in SQL, such as 'alter' and 'double'. Rather than substituting a completely different name, an underscore is added to the name.

The following table shows the properties found in the other default nodes in a MusicSQL database. Most can only be accessed from one or two types of Hub object via its '`node`' method. Some of these nodes do not contain any selectable information, but can act as true/false tests. Their '`is_null()`' method returns this test (a check as to whether there is any value for that Hub). The 'True/False' column below indicates whether this test is meaningful for the node.

Node	Related Hubs	True/False	Properties
files	Part, Moment, Event		`path, work_title, work_number, movement_number, movement_title, rights, encoding_date, encoder, source, divisions`
measures	Event		`number, implicit, non_controlling, width`
attributes	Event		`fifths, mode, beats, beat_type, instruments, sign, line, clef_octave_change, staves, directive, symbol, diatonic, chromatic, double_`
barlines	Event	✔	`style, location`
slurs	Notehead	✔	*no selectable properties*
wedges	Event	✔	`type`
beams	Notehead		`beam1, beam2, beam3, beam4, beam5, beam6`
directions	Event	✔	`type, value`

Additional information nodes can be added via modules. Here are the properties of the module nodes that ship with the current version of MusicSQL:

Node	Related Hubs	True/False	Properties
cadence	Moment	✔	*no selectable properties*
contrarymotion	Note	✔	*no selectable properties*
doubledparts	Moment	✔	*no selectable properties*
hornfifth	Moment	✔	`ideal_vl`
lownote	Moment, Note	✔(Note)	*no selectable properties*
nht	Moment, Note	✔(Note)	`quality_wo_nht`
notecount	Moment		`notecount`
onsetcount	Moment		`onsetcount`
quality	Moment		`quality, root, inversion`
topnote	Moment, Note	✔(Note)	*no selectable properties*
upbeat	Moment	✔	*no selectable properties*

B Object methods

Queries in MusicSQL are constructed outward via relationships from existing objects. The construction process is conducted via Hub object methods that create new objects. (The Hub objects are Parts, Notes, Noteheads, Moments, Events, and Simultaneities.) The following tables list the constructor methods for the Hubs objects. Optional arguments are given in brackets.

B.1 Part constructor methods.

Method	Result	Description
`add_first_note([details])`	Note	Creates the first Note of a query.[a]
`add_moment()`	Moment	Creates the first Moment of a query.[a]
`add_note(Moment [, details])`	Note	Adds a Note to a particular Event.
`event(Moment)`	Event	Returns the Event in this Part at a given Moment.

[a] This method can only be called while there are no existing Notes or Moments.

B.2 Note constructor methods.

Method	Result	Description
`notehead()`	Notehead	The first Notehead connected to this Note.
`part()`	Part	This Note's Part.
`start_moment()`	Moment	The Moment of this Note's onset.
`start_event()`	Event	The Event this Note begins at.
`add_simultaneity()`	Simultaneity	Creates a Simultaneity attached to this Note
`newevent_note([details])`	Note	Creates another Note at the same start Event (creates a chord).
`add_next_note([details])`	Note	Creates the Note that comes next in this Part (in this voice).
`add_next_beat([details])`	Note	Creates the Note that starts at the next beat.
`add_next_note_or_beat([details])`	Note	Creates a Notes that either comes directly after or starts at the next beat.
`add_next_semit([details])`	Note	Creates the next Note with a different `semit` property.
`add_note_sequence(details [,details ...])`	Notes	Calls `add_next_note()` for each detail.
`add_diatonic_interval_sequence(interval [, interval ...])`	Notes	Like `add_note_sequence`; see the `set_contour` method
`add_contour_sequence(contour [, contour ...])`	Notes	Like `add_note_sequence`; see the `set_diatonic_interval` method
`add_previous_note()`	Note	see `add_next_note`
`add_previous_beat()`	Note	see `add_next_beat`
`add_previous_note_or_beat()`	Note	see `add_next_note_or_beat`
`add_previous_semit()`	Note	see `add_next_semit`

B.3 Notehead constructor methods.

Method	Result	Description
`add_next_tied_notehead()`	Notehead	Creates a Notehead that is tied to this Notehead

B.4 Moment constructor methods.

Method	Result	Description
`add_note(Part [,details])`	Note	Adds a note to a particular Event.
`add_later_moment()`	Moment	Creates a new moment at some point later in the movement.
`add_earlier_moment()`	Moment	Creates a new moment at some point earlier in the movement.
`event(Part)`	Event	Returns the Event in this Moment in the given Part.

B.5 Event constructor methods.

Method	Result	Description
`part()`	Part	Returns the Part this Event is in.
`moment()`	Moment	Returns the Moment this Event is in.

B.6 Simultaneity constructor methods.

Method	Result	Description
`add_note(Part [, details])`	Note	Creates a new Note and attaches it to this Simultaneity.

Hub objects have a number of other methods which are useful for constructing relationships or setting details of the objects. Below is a table of the methods that have not been mentioned to this point.

B.7 Miscellaneous methods

Hub	Method	Description
Note	`set_note_details(details)`	A method to add details to a Note after it has been created.
	`set_diatonic_interval( Note2, interval [,dir])`	Adds a constraint between this Note and Note2: their `concert_step` properties are an `interval` apart.[a]
	`set_chromatic_interval( Note2, interval [,dir])`	Adds a constraint between this Note and Note2: their `semit` properties are an `interval` apart.[a]
	`set_contour(Note2, contour [,dir])`	Adds a constraint between this Note and Note2: the contour to the later note is up (`'^'`) or down (`'v'`) or none (`'-'`).[a]
Notehead	`set_slur(Notehead2)`	Adds a constraint that this Notehead starts a slur and Notehead2 ends it.
Moment	`measure([Part])`	Returns the measure node of an Event (the one in the original Part, by default) in one step.
	`attributes([Part])`	Like `measure()`, but returns the attributes node.
Event	`set_wedge(Event2)`	Adds a constraint that this Event starts a wedge and Event2 ends it.
Simultaneity	`attach_note(Note)`	Attaches a Note to this Simultaneity.
All Hubs	`same_as(Hub2)`	Returns an expression that tests whether this Hub and Hub2 are actually the same object.
	`add_constraint(expression)`	Makes an expression a requirement for the query. The expression must relate to this Hub.
	`node('node')`	Returns a Node attached to this Hub. (See Appendix A.)
All Nodes	`is_null()`	Returns an expression that tests whether this Node has a value for its Hub.
	`select('property' [,'property', ...])`	*See section 5.1.2.*
	`select_alias('property', 'alias')`	*See section 5.1.2.*
	`select_all()`	*See section 5.1.2.*

[a] Note2 is assumed to be later unless the optional `dir` argument is set to `'previous'`.

B.8 Query object methods

Method	Description
`part()`	Returns a Part. Callable any number of times.
`conditional()`	Returns an empty Conditional object.
`select_expression( expression, alias)`	Adds the expression to the output with the given alias.
`select_all()`	Calls the `select_all()` method on all Nodes (and Hubs)
`sqlfunction('function')`	Returns a callable function that provides the requested SQL function[a]
`print_run()`	Executes the query and prints the output.

[a] Some common functions are ROUND, MAX, and COUNT. A partial list of SQL functions is available at:

`http://www.w3schools.com/SQl/sql_functions.asp`

C A note regarding Humdrum

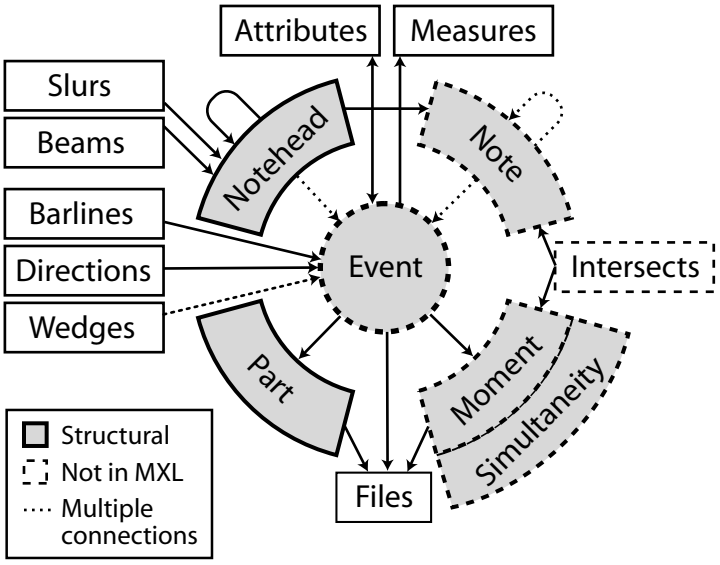
Anyone who has used Humdrum will see a similarity of purpose with MusicSQL. Humdrum is a powerful, lightweight (in the best sense), multipurpose music search tool, and it is the best choice for many tasks. The Humdrum representation is efficient and extremely useful for scripting, and nothing will replace it.

One way to think about the purpose of MusicSQL is that it tries to make the things that are hard in Humdrum easier. It is difficult to search across a number of dimensions simultaneously in Humdrum, but it is easy in MusicSQL. It is hard to search across a number of files simultaneously in Humdrum, but MusicSQL does it by default. It is difficult to imagine an object-based search GUI built on Humdrum, but it is implicit in the structure of MusicSQL.

Each research tool has its purpose, and there are many things that Humdrum does well that MusicSQL will likely never do. Humdrum's representation is flexible and can represent anything, but MusicSQL is built on MusicXML and is locked into the Common Music Notation standard. Humdrum can restructure data and integrate into scripting languages very easily, whereas MusicSQL is built on an SQL database with fixed tables and relationships.

With an array of software available, researchers can settle on the system that works best for the project at hand and their own computer skills. To each according to his need, from the tools according to their abilities.

D The basic database design



If you have any bug reports or suggestions, please email the author:
Bret Aarden <bret.aarden at gmail.com>