CSC 459: Database Management Systems – Final Project

Jacob M. Peck

May 9, 2012

Abstract

A high-level description of a music information database is provided, along with an analysis of functional dependencies, implementation notes, and full source code.

Contents

1 Description

This project is a music information database, consisting of information stored in the ID3 tags of my music library. These are stored as a MusicObject object which holds several fields, mapped to by a unique universal identifier, in key-value pairs. The data is stored on multiple machines in a redundant fashion, and they are all polled simultaneously, working out consensus between nodes.

1.1 Users

The users of this system would be anyone interested in this type of information. Users would be able to ask the database for questions involving aggregation, such as the average track length in seconds of songs by a particular artist.

1.2 Types of data

This system has only one type of data: a MusicObject. However, the MusicObject itself is a compound field, consisting of several character strings and integers, as follows:

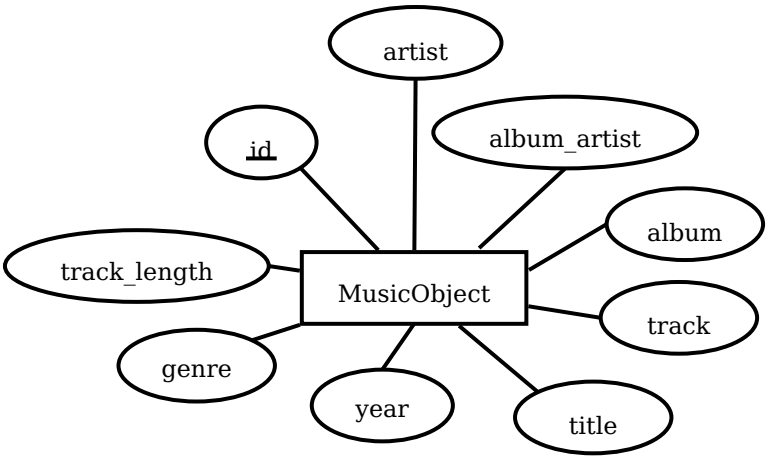
field	**type**
id	string
artist	string
album_artist	string
album	string
track	integer
title	string
year	integer
genre	string
track_length	integer

1.3 Use cases

A common use case for this system would be to query for either information about a particular song, or for aggregate data about a particular subset of songs. The information held in the database lends itself well to map-reduce style queries.

2 ER Diagram

Below is an entity-relationship model for the data in this system. You will notice that this is rather sparse, as the entirety of the system contains of a single type of entity.



3 Functional dependency analysis

What follows is an analysis of the functional dependencies as present in the data model. As the entire system consists of a single table, the functional dependencies are rather simplistic.

3.1 Functional dependencies

- $id \rightarrow id, artist, album\_artist, album, track, title, year, genre, track\_length$
- $artist \rightarrow artist$
- $album\_artist \rightarrow album\_artist$
- $album \rightarrow album$
- $track \rightarrow track$
- $title \rightarrow title$
- $year \rightarrow year$
- $genre \rightarrow genre$
- $track\_length \rightarrow track\_lenght$
- $artist, album, track \rightarrow album, artist, track, title$

3.2 Normal form of the data

Analyzing this data with the above functional dependencies, it can be shown that the data is in 1NF, as there are zero multivalued attributes. The data is also in 3NF, as every functional dependency is either trivial, a superkey, or a prime attribute. The data is *not* in BCNF, however, as the final functional dependency ($artist, album, track \rightarrow artist, album, track, title$) is not either a trivial functional dependency or a superkey.

The highest normal form that is satisfied by this data is 3NF for the above stated reasons.

4 Description of the schema

The schema is fairly straight forward, as previously mentioned. No field is allowed to be `null`, and the id field must be unique.

field	type
id	string
artist	string
album_artist	string
album	string
track	integer
title	string
year	integer
genre	string
track_length	integer

5 Implementation

The implementation of this system was accomplished by a team of 4 members: myself, Joe Mirizio, Karl Miller, and Nate Hemmes. It was written in Java 7 (source attached, see section 7).

5.1 Architecture

The system is composed of a client and several server nodes. Each server node is identical in it's behavior, but potentially different in the data it has stored.

5.1.1 Server nodes

The server nodes are capable of being queried for records matching a particular combination of traits, and will return a collection of all of the elements which match the specified query. The nodes are redundant and will communicate with each other in order to come to a consensus before providing data to the client.

5.1.2 Client

The client node queries the server nodes, and then applies a map/reduce style aggregation on the returned collection.

5.2 Example data

Here is an example record (split over two lines for readability):

id	artist	album_artist	album	track
1344f4d7-ca00-4a6d-9d35-de940a6beda1	Fear Factory	Fear Factory	Digimortal	4

title	year	genre	track_length
No One	2001	Rock/Pop	218

6 Interface

The interface will be the client node, which will have a command-line interface with which the user can query the database.

6.1 Example query

An example query might look like this:

`GET 1 ARTIST='Foo Fighters', YEAR='2001'` Which would return a collection of at most 1 MusicObject from which artist exactly equals the string 'Foo Fighters', and the year exactly equals '2001'.

Another example might be as follows:

`GET ALL GENRE='Death Metal'` Which would return a collection of all the MusicObjects which have 'Death Metal' as their genre string.

From this, the client will perform map/reduce functions, such as the following example in pseudocode:

```
// query
dm = query("GET ALL GENRE='Death Metal'")

// map
for each x in dm:
  replace x with x.artist
end

// reduce
count = 0
for each x in dm:
  if x == "In Flames"
    count++
  end
end

// final result (count of all songs with genre 'Death Metal' with artist 'In Flames')
print count
```

7 Code

7.1 Client.java

```
// Client class

// Jacob Peck
// Joe Mirizio

import java.util.*;
import java.io.*;
import java.net.*;

interface MapFunction<T, U> {
  U operate(T val);
}

interface ReduceFunction<U, V> {
  V operate(U val);
}

class MapReduce<T, U, W, V> {
  public MapFunction<T,U> map;
  public ReduceFunction<W,V> reduce;
```

```
22    public List<U> callMap(List<T> val) {
        List<U> alist = new ArrayList<U>();
        for (T t : val) {
          alist.add(this.map.operate(t));
        }
27      return alist;
      }

      public V callReduce(W val) { return this.reduce.operate(val); }
    }
32
    public class Client {
      private ArrayList<InetSocketAddress> servers = new ArrayList<InetSocketAddress>();
      private final String SERVER_LIST_FILE_NAME = "servers.txt";
      private final int CONNECTION_TIMEOUT = 2000;
37
      public Client() {
        try {
          File f = new File(SERVER_LIST_FILE_NAME);
          Scanner sc = new Scanner(f);
42        while(sc.hasNextLine()) {
            Scanner tokenizer = new Scanner(sc.nextLine());
            servers.add(new InetSocketAddress(tokenizer.next(), Integer.parseInt(tokenizer.next())));
          }
        } catch (Exception ex) {
47        System.out.println("Something went wrong:");
          ex.printStackTrace();
        }
      }

52    public static void main(String[] args) {
        // set up connections
        Client c = new Client();

        // send out requests
57      // store
        //System.out.println("Adding a new MusicObject");
        //MusicObject mobj = new MusicObject(UUID.randomUUID().toString(), "Some Artist", "Some Album Artist",
        //                                   "Some Album", 3, "Some Title", 2012, "Some Genre", 323);
        //c.store(mobj);
62
        try{Thread.sleep(500);} catch (Exception ex){}

        // kill
        /*
67      System.out.println("\n\nKilling a server...");
        c.kill();
        */

        try{Thread.sleep(500);} catch (Exception ex){}
72
        // map reduce 1
        MapReduce<MusicObject, Integer, List<Integer>, Integer> mr1 = new MapReduce<MusicObject, Integer, List<
            Integer>, Integer>();
        mr1.map = new MapFunction<MusicObject, Integer>() {
          public Integer operate(MusicObject mobj) {return mobj.getTrackLength();}
77      };
        mr1.reduce = new ReduceFunction<List<Integer>, Integer>() {
          public Integer operate(List<Integer> list) {
            int total = list.size();
            if(total == 0) return 0;
82          int accum = 0;
```

```
        for(Integer i : list) accum += i;
        return accum / total;
      }
    };
    System.out.println("\n\nMap/Reduce #1: average length in seconds of a song by Opeth:");
    HashMap<String, String> filter = new HashMap<String, String>();
    filter.put("ARTIST", "Opeth");
    ArrayList<MusicObject> alist = null;
    while(alist == null) {
      try{
        alist = c.query("ALL", filter);
      } catch (Exception ex) {
        alist = null;
        continue;
      }
    }
    // map
    List<Integer> track_lengths = mr1.callMap(alist);
    // reduce
    Integer avg_length = mr1.callReduce(track_lengths);
    System.out.println("Average length is: " + avg_length);
    System.out.println("INFO: " + alist.size() + " entries.");

    try{Thread.sleep(500);} catch (Exception ex){}

    // map reduce 2
    MapReduce<MusicObject, Integer, List<Integer>, Integer> mr2 = new MapReduce<MusicObject, Integer, List<
        Integer>, Integer>();
    mr2.map = new MapFunction<MusicObject, Integer>() {
      public Integer operate(MusicObject mobj) {return mobj.getYear();}
    };
    mr2.reduce = new ReduceFunction<List<Integer>, Integer>() {
      public Integer operate(List<Integer> list) {
        HashMap<Integer, Integer> accum = new HashMap<Integer, Integer>();
        for(Integer i : list) {
          if(accum.get(i) == null) accum.put(i, 1);
          else accum.put(i, accum.get(i) + 1);
        }
        Integer median = 0;
        for(Map.Entry<Integer, Integer> e : accum.entrySet()) {
          if(median == 0) median = e.getKey();
          if(e.getValue() > accum.get(median)) median = e.getKey();
        }
        return median;
      }
    };
    System.out.println("\n\nMap/Reduce #2: most active recording year for the Foo Fighters:");
    filter = new HashMap<String, String>();
    filter.put("ARTIST", "Foo Fighters");
    alist = null;
    while(alist == null) {
      try{
        alist = c.query("ALL", filter);
      } catch (Exception ex) {
        alist = null;
        continue;
      }
    }
    // map
    List<Integer> years = mr2.callMap(alist);
    // reduce
    Integer median_year = mr2.callReduce(years);
```

```
    System.out.println("Median year is: " + median_year);
    System.out.println("INFO: " + alist.size() + " entries.");

    try{Thread.sleep(500);} catch (Exception ex){}

    // map reduce 3
    MapReduce<MusicObject, Integer, List<MusicObject>, MusicObject> mr3 = new MapReduce<MusicObject, Integer
        , List<MusicObject>, MusicObject>();
    mr3.reduce = new ReduceFunction<List<MusicObject>, MusicObject>() {
      public MusicObject operate(List<MusicObject> list) {
        MusicObject retval = null;
        for(MusicObject m : list) {
          if(retval == null) retval = m;
          else if(m.getTrackLength() > retval.getTrackLength())
            retval = m;
        }
        return retval;
      }
    };
    System.out.println("\n\nMap/Reduce #3: Longest track on St. Elsewhere by Gnarls Barkley");
    filter = new HashMap<String, String>();
    filter.put("ARTIST", "Gnarls Barkley");
    filter.put("ALBUM", "St. Elsewhere");
    alist = null;
    while(alist == null) {
      try{
        alist = c.query("ALL", filter);
      } catch (Exception ex) {
        alist = null;
        continue;
      }
    }
    // reduce
    MusicObject longest = mr3.callReduce(alist);
    System.out.println("Longest track is: " + longest);
    System.out.println("INFO: " + alist.size() + " entries.");

    try{Thread.sleep(500);} catch (Exception ex){}

    // kill
    System.out.println("\n\nKilling a server...");
    c.kill();

    try{Thread.sleep(500);} catch (Exception ex){}

    // map reduce 3 again
    System.out.println("\n\nMap/Reduce #3 (after kill): Longest track on St. Elsewhere by Gnarls Barkley");
    filter = new HashMap<String, String>();
    filter.put("ARTIST", "Gnarls Barkley");
    filter.put("ALBUM", "St. Elsewhere");
    alist = null;
    while(alist == null) {
      try{
        alist = c.query("ALL", filter);
      } catch (Exception ex) {
        alist = null;
        continue;
      }
    }
    // reduce
    longest = mr3.callReduce(alist);
    System.out.println("Longest track is: " + longest);
```

```
    System.out.println("INFO: " + alist.size() + " entries.");
  }

  // query
  public ArrayList<MusicObject> query(String limit, HashMap<String, String> filter) {
    Socket req = null;
    while(req == null) {
      try {
        InetSocketAddress sa = servers.get((int)Math.floor(Math.random() * servers.size()));
        System.out.println("Trying address " + sa);
        req = new Socket();
        req.connect(sa, CONNECTION_TIMEOUT);
      } catch (Exception ex) {
        req = null;
      }
    }
    ObjectInputStream ois = null;
    //PrintWriter pwo = null;
    DataOutputStream dos = null;
    try {
      dos = new DataOutputStream(req.getOutputStream());
      //pwo = new PrintWriter(req.getOutputStream());
    } catch (EOFException ex) {
      return null;
    } catch (Exception ex) {
      ex.printStackTrace();
    }

    // construct query
    StringBuilder sb = new StringBuilder("GET ");
    sb.append(limit + " ");
    for(Map.Entry<String, String> e : filter.entrySet()) {
      sb.append(e.getKey() + "='" + e.getValue() + "', ");
    }
    String query = sb.toString();
    query = query.substring(0, query.length() - 2);
    System.out.println(query);

    // send query
    try{dos.writeUTF(query);} catch(Exception ex) {}

    // read in response
    ArrayList<MusicObject> retval = new ArrayList<MusicObject>();
    try {
      ois = new ObjectInputStream(req.getInputStream());
      Object ret = ois.readObject();

      @SuppressWarnings("unchecked")
      ArrayList<Object> ret2 = (ArrayList<Object>) ret;

      for(Object o : ret2) {
        retval.add((MusicObject)o);
      }
    } catch (EOFException ex) {
      return null;
    } catch (Exception ex) {
      ex.printStackTrace();
    }

    // return it
    try{req.close();} catch(Exception ex) {}
```

```
    return retval;
  }

  // store
  public void store(MusicObject mobj) {
    Socket req = null;
    while(req == null) {
      try {
        InetSocketAddress sa = servers.get((int)Math.floor(Math.random() * servers.size()));
        req = new Socket();
        req.connect(sa, CONNECTION_TIMEOUT);
      } catch (Exception ex) {
        req = null;
      }
    }
    //PrintWriter pwo = null;
    DataOutputStream dos = null;
    ObjectInputStream ois = null;
    try {
      //pwo = new PrintWriter(req.getOutputStream());
      dos = new DataOutputStream(req.getOutputStream());
      ois = new ObjectInputStream(req.getInputStream());
    } catch (Exception ex) {
      ex.printStackTrace();
    }

    // construct query
    StringBuilder sb = new StringBuilder("PUT ");
    sb.append((System.nanoTime() + (Math.floor(Math.random() * 1000000))) + ", ");
    sb.append(mobj.getId() + ", ");
    sb.append(mobj.getArtist() + ", ");
    sb.append(mobj.getAlbumArtist() + ", ");
    sb.append(mobj.getAlbum() + ", ");
    sb.append(mobj.getTrack() + ", ");
    sb.append(mobj.getTitle() + ", ");
    sb.append(mobj.getYear() + ", ");
    sb.append(mobj.getGenre() + ", ");
    sb.append(mobj.getTrackLength());
    String query = sb.toString();
    System.out.println(query);

    // send query
    //pwo.println(query);
    try{
      dos.writeUTF(query);
      ois.readObject();
      req.close();
    } catch(Exception ex) {}
  }

  // kill
  public void kill() {
    Socket req = null;
    while(req == null) {
      try {
        InetSocketAddress sa = servers.get((int)Math.floor(Math.random() * servers.size()));
        req = new Socket();
        req.connect(sa, CONNECTION_TIMEOUT);
        System.out.println("Killing node: " + sa);
      } catch (Exception ex) {
        req = null;
      }
```

```
    }
    //PrintWriter pwo = null;
    DataOutputStream dos = null;
    try {
      dos = new DataOutputStream(req.getOutputStream());
    } catch (Exception ex) {
      ex.printStackTrace();
    }

    // construct query
    String query = "KILL";

    // send query
    //pwo.println(query);
    try{
      dos.writeUTF(query);
      req.close();
    } catch(Exception ex) {}
  }
}
```

Client.java

7.2 MusicObject.java

```
// MusicObject class

// Jacob Peck
// Karl Miller (1 method)

import java.io.*;
import java.util.*;

enum Mode {
  MODE_NONE, MODE_ARTIST, MODE_ALBUM_ARTIST, MODE_ALBUM, MODE_TRACK,
  MODE_TITLE, MODE_YEAR, MODE_GENRE, MODE_TRACK_LENGTH, MODE_SONG
}

public class MusicObject implements Serializable, Comparable {

  private String id;
  private String artist;
  private String album_artist;
  private String album;
  private int track;
  private String title;
  private int year;
  private String genre;
  private int track_length;

  public String getId() {return id;}
  public String getArtist() {return artist;}
  public String getAlbumArtist() {return album_artist;}
  public String getAlbum() {return album;}
  public int getTrack() {return track;}
  public String getTitle() {return title;}
  public int getYear() {return year;}
  public String getGenre() {return genre;}
  public int getTrackLength() {return track_length;}
```

```
  // setters... needed for factory method :(
37 public void setId(String id) {this.id = id;}
   public void setArtist(String artist) {this.artist = artist;}
   public void setAlbumArtist(String album_artist) {this.album_artist = album_artist;}
   public void setAlbum(String album) {this.album = album;}
   public void setTrack(int track) {this.track = track;}
42 public void setTitle(String title) {this.title = title;}
   public void setYear(int year) {this.year = year;}
   public void setGenre(String genre) {this.genre = genre;}
   public void setTrackLength(int track_length) {this.track_length = track_length;}

47 public static MusicObject mobj;

   private MusicObject(){}

   public MusicObject(String id, String artist, String album_artist, String album,
52                    int track, String title, int year, String genre, int track_length) {
     this.id = id;
     this.artist = artist;
     this.album_artist = album_artist;
     this.album = album;
57   this.track = track;
     this.title = title;
     this.year = year;
     this.genre = genre;
     this.track_length = track_length;
62 }

   public String toString() {
     StringBuilder sb = new StringBuilder();
     sb.append("<mobj " + id + "\n");
67   sb.append("  artist: " + artist + "\n");
     sb.append("  album_artist: " + album_artist + "\n");
     sb.append("  album: " + album + "\n");
     sb.append("  track: " + track + "\n");
     sb.append("  title: " + title + "\n");
72   sb.append("  year: " + year + "\n");
     sb.append("  genre: " + genre + "\n");
     sb.append("  track_length: " + track_length + "\n");
     sb.append(">");
     return sb.toString();
77 }

   public static MusicObject buildFromFile(File f, String id) {
     MusicObject.mobj = new MusicObject();
     Scanner sc = null;
82   try {
       sc = new Scanner(f);
       while(sc.hasNextLine()) {
         String line = sc.nextLine().trim();
         if(line.equals("<?xml ?>")) continue;
87       if(line.equals("<song>")) continue;
         if(line.equals("</song>")) continue;
         if(line.startsWith("<artist>")) {
           line = line.replaceFirst("<artist>", "");
           line = line.replaceFirst("</artist>", "");
92         mobj.setArtist(line);
           continue;
         }
         if(line.startsWith("<album_artist>")) {
           line = line.replaceFirst("<album_artist>", "");
97         line = line.replaceFirst("</album_artist>", "");
```

```
          mobj.setAlbumArtist(line);
          continue;
        }
        if(line.startsWith("<album>")) {
          line = line.replaceFirst("<album>", "");
          line = line.replaceFirst("</album>", "");
          mobj.setAlbum(line);
          continue;
        }
        if(line.startsWith("<track>")) {
          line = line.replaceFirst("<track>", "");
          line = line.replaceFirst("</track>", "");
          mobj.setTrack(Integer.parseInt(line));
          continue;
        }
        if(line.startsWith("<title>")) {
          line = line.replaceFirst("<title>", "");
          line = line.replaceFirst("</title>", "");
          mobj.setTitle(line);
          continue;
        }
        if(line.startsWith("<year>")) {
          line = line.replaceFirst("<year>", "");
          line = line.replaceFirst("</year>", "");
          mobj.setYear(Integer.parseInt(line));
          continue;
        }
        if(line.startsWith("<genre>")) {
          line = line.replaceFirst("<genre>", "");
          line = line.replaceFirst("</genre>", "");
          mobj.setGenre(line);
          continue;
        }
        if(line.startsWith("<track_length>")) {
          line = line.replaceFirst("<track_length>", "");
          line = line.replaceFirst("</track_length>", "");
          mobj.setTrackLength(Integer.parseInt(line));
          continue;
        }
      }
    } catch (Exception ex) {
      return MusicObject.mobj;
    }
    sc.close();
    MusicObject.mobj.setId(id);
    return MusicObject.mobj;
  }

  public boolean validate() {
    //System.out.println(this);
    if(this.id == null) return false;
    if(this.artist == null) return false;
    if(this.album_artist == null) return false;
    if(this.album == null) return false;
    if(this.track == 0) return false;
    if(this.title == null) return false;
    if(this.year == 0) return false;
    if(this.genre == null) return false;
    if(this.track_length == 0) return false;
    return true;
  }
```

```
public boolean matches(HashMap<String, String> filter) {
  boolean match = true;
  for(Map.Entry<String, String> e : filter.entrySet()) {
    //System.out.println(this.toString());
    switch(e.getKey()) {
      case "ARTIST":
        if(this.artist != null) {
          //System.out.println(e.getValue() + " ? " + this.artist);
          match = (this.artist.equals(e.getValue()));
        } else
          match = false;
        break;
      case "ALBUM_ARTIST":
        if(this.album_artist != null)
          match = (this.album_artist.equals(e.getValue()));
        else
          match = false;
        break;
      case "ALBUM":
        if(this.artist != null)
          match = (this.album.equals(e.getValue()));
        else
          match = false;
        break;
      case "TRACK":
        match = (Integer.toString(this.track).equals(e.getValue()));
        break;
      case "TITLE":
        if(this.title != null)
          match = (this.title.equals(e.getValue()));
        else
          match = false;
        break;
      case "YEAR":
        match = (Integer.toString(this.year).equals(e.getValue()));
        break;
      case "GENRE":
        if(this.genre != null)
          match = (this.genre.equals(e.getValue()));
        else
          match = false;
        break;
      case "TRACK_LENGTH":
        match = (Integer.toString(this.track_length).equals(e.getValue()));
        break;
      default:
        break;
    }
    if(!match) break;
  }
  return match;
}

/**
 * This method compares two MusicObjects. The method only compares the
 * equality of the following fields:<br />
 *        String id;<br />
 *        String artist;<br />
 *        String album_artist;<br />
 *        String album;<br />
 *        int track;<br />
 *        String title;<br />
```

```
222     *    &amp;nbsp;&amp;nbsp;&amp;nbsp;&amp;nbsp;int year;<br />
        *    &amp;nbsp;&amp;nbsp;&amp;nbsp;&amp;nbsp;String genre;<br />
        *    &amp;nbsp;&amp;nbsp;&amp;nbsp;&amp;nbsp;int track_length;<br />
        * Upon comparing the values in the specified fields, this method returns
        * an integer based value, signaling what fields are equal and what fields
227     * are not. The returned value should be considered a binary value, and the
        * bits set tell which fields are not equal. For example:<br />
        * <br />
        * 19 = 0x13: signals that the fields, id, artist, and track are not equal
        * 0 = 0x0: signals all fields are equal
232     * -1 = 0xFFFFFFFF: signals that the object provided is not a MusicObject
        * @param o the object that will be compared to this
        * @return 0 if objects are equal, -1 if o is not a MusicObject, or the
        *    following binary values or'd together otherwise:
        *      0x1: id is not equal
237     *      0x2: artist is not equal
        *      0x4: album_artist is not equal
        *      0x8: album is not equal
        *      0x10: track is not equal
        *      0x20: title is not equal
242     *      0x40: year is not equal
        *      0x80: genre is not equal
        *      0x100: track_length is not equal
        */
       public int compareTo(Object o){
247      if(!(o instanceof MusicObject)){
           return -1;
         }
         int returnValue = 0;
         MusicObject m = (MusicObject)o;
252      if(!m.id.equals(id)){
           returnValue |= 0x1;
         }
         if(!m.artist.equals(artist)){
           returnValue |= 0x2;
257      }
         if(!m.album_artist.equals(album_artist)){
           returnValue |= 0x4;
         }
         if(!m.album.equals(album)){
262        returnValue |= 0x8;
         }
         if(m.track != track){
           returnValue |= 0x10;
         }
267      if(!m.title.equals(title)){
           returnValue |= 0x20;
         }
         if(m.year != year){
           returnValue |= 0x40;
272      }
         if(!m.genre.equals(genre)){
           returnValue |= 0x80;
         }
         if(m.track_length != track_length){
277        returnValue |= 0x100;
         }
         return returnValue;
       }

282    public boolean equals(Object o) {
         return this.compareTo(o) == 0;
```

```
  }
}
```

MusicObject.java

7.3 Node.java

```
/**
 * CSC 445
 * Project 3
 */

// Joe Mirizio
// Jacob Peck

import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.Serializable;
import java.io.File;
import java.util.Map;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Collection;
import java.util.concurrent.*;

/**
 * A representation of a persistent data node.
 */
public class Node implements Serializable {

  /**
   * The filename for the data persistence.
   */
  protected static final String FILENAME = "data.bin";

  /**
   * The data stored in the node.
   */
  protected ConcurrentHashMap<String, MusicObject> data;


  /**
   * Constructor.
   */
  public Node() {
    this.data = new ConcurrentHashMap<String, MusicObject>();
  }

  /**
   * Add a MusicObject to the node.
   * @param song The song to be added.
   */
  public void addMusicObject(MusicObject song) {
    this.data.put(song.getId(), song);
  }

  /**
```

```
   * Persists the data to the file.
   */
  protected void writeToFile() {
    try {
      FileOutputStream fos = new FileOutputStream(FILENAME);
      ObjectOutputStream oos = new ObjectOutputStream(fos);

      oos.writeObject(this.data);

      oos.close();
    } catch (IOException e) { e.printStackTrace(); }
  }

  /**
   * Reinitializes the node from the file.
   */
  @SuppressWarnings("unchecked")
  protected void readFromFile() {
    try {
      FileInputStream fis = new FileInputStream(FILENAME);
      ObjectInputStream ois = new ObjectInputStream(fis);

      this.data = (ConcurrentHashMap<String, MusicObject>)ois.readObject();

      ois.close();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }

  /**
   * Query for data.
   * filter - a HashMap<String,String> for matching against.
   */
  protected Collection<MusicObject> query(HashMap<String, String> filter) {
    Collection<MusicObject> resultset = new ArrayList<MusicObject>();
    for(Map.Entry<String, MusicObject> e : data.entrySet()) {
      MusicObject m = e.getValue();
      if(m.matches(filter)) resultset.add(m);
    }
    return resultset;
  }

  protected HashMap<String, String> parseRequest(String request) {
    HashMap<String, String> filter = new HashMap<String, String>();

    boolean looking = false;
    boolean master = false;
    boolean seekAll = false;

    int begin = 0;
    int filled = 0;
    int saughtSize = 0;//this is how many responsed are desired

    String[] parsedRequest = new String[10];
    String singleQ = "\'";

    for (int c = 0; c < request.length(); c++) {
      char ch = request.charAt(c);
      if (ch == ',') {
```

```
        if (begin < (c - 1)) {
          parsedRequest[filled] = request.substring(begin, c);
          begin = c + 1;
          filled = filled + 1;
        }
        while (c < (request.length() - 1) && request.charAt(c + 1) == ' ') {
          c = c + 1;
          begin = c + 1;
        }
        if (c >= (request.length() - 1)) {
          break;
        }
      } else if (ch == '\'') {
        c = c + 1;
        while (c < (request.length()) && request.charAt(c) != '\'') {
          c = c + 1;
        }
        if (c == (request.length() - 1)) {
          parsedRequest[filled] = request.substring(begin, c + 1);
          filled = filled + 1;
        }
      } else if (ch == ' ') {
        if (begin < (c)) {
          parsedRequest[filled] = request.substring(begin, c);
          begin = c + 1;
          filled = filled + 1;
        }
      }
    } //done building the array of commands

    if (parsedRequest[0].compareToIgnoreCase("get") == 0) {
      if (parsedRequest[1].compareToIgnoreCase("all") == 0) {
        master = true;
        seekAll = true;
      } else if (parsedRequest[1].compareTo("n") == 0) {
        master = false;
        seekAll = true;
      } else {
        try {
          int x = Integer.parseInt(parsedRequest[1]);
          saughtSize = x;
        } catch (NumberFormatException nfe) {
          System.out.println("Number Format Exception: " + nfe.getMessage());
        }
      }

      String[] getFields = null;

      for (int pointer = 2; pointer < filled; pointer++) {//identifies the fields and gets the objects with
          those clasifications
        System.out.println("parsedRequest at " + pointer + " is " + parsedRequest[pointer]);
        getFields = parsedRequest[pointer].split("=");
        if (getFields[0].compareToIgnoreCase("artist") == 0) {
          filter.put("ARTIST", getFields[1].substring(1, getFields[1].length()-1));
        } else if ((getFields[0].compareToIgnoreCase("albumartist") == 0) || (getFields[0].
            compareToIgnoreCase("album_artist") == 0)) {
          filter.put("ALBUM_ARTIST", getFields[1].substring(1, getFields[1].length()-1));
        } else if (getFields[0].compareToIgnoreCase("album") == 0) {
          filter.put("ALBUM", getFields[1].substring(1, getFields[1].length()-1));
        } else if (getFields[0].compareToIgnoreCase("track") == 0) {
          filter.put("TRACK", getFields[1].substring(1, getFields[1].length()-1));
        } else if (getFields[0].compareToIgnoreCase("title") == 0) {
```

```
          filter.put("TITLE", getFields[1].substring(1, getFields[1].length()-1));
        } else if (getFields[0].compareToIgnoreCase("year") == 0) {
          filter.put("YEAR", getFields[1].substring(1, getFields[1].length()-1));
        } else if (getFields[0].compareToIgnoreCase("genre") == 0) {
          filter.put("GENRE", getFields[1].substring(1, getFields[1].length()-1));
        } else if ((getFields[0].compareToIgnoreCase("tracklength") == 0) || (getFields[0].
            compareToIgnoreCase("track_length") == 0)) {
          filter.put("TRACK_LENGTH", getFields[1].substring(1, getFields[1].length()-1));
        } else {
          //Improper field label;
        }
      }
    }
    return filter;
  }

  /**
   * main - initialize node and save to file
   */
  public static void main(String[] args) {
    Node node = new Node();
    node.loadFiles("data");
    node.writeToFile();
  }

  /**
   * Loads all of the files in dataDir into the node
   */
  private void loadFiles(String dataDir) {
    File[] files = (new File(dataDir)).listFiles();
    for(int i = 0; i < files.length; i++) {
      try {
        File f = files[i];
        String id = f.getName().substring(0, f.getName().lastIndexOf('.'));
        //System.out.println("Adding MusicObject: " + id);
        MusicObject mobj = MusicObject.buildFromFile(f, id);
        if(mobj.validate()){
          this.addMusicObject(mobj);
          System.out.println(mobj);
        }
      } catch (Exception ex) {continue;}
    }
  }
}
```

Node.java

7.4 NodeServer.java

```
/**
 * CSC 445
 * Project 3
 */

// Joe Mirizio
// Jacob Peck
// Nate Hemmes (1 method)

import java.io.*;
import java.net.*;
```

```java
import java.nio.*;
import java.util.Scanner;
import java.util.List;
import java.util.Collection;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.concurrent.*;

public class NodeServer extends Thread {

  public static final int DEFAULT_PORT = 2697;
  public static final String DEFAULT_SERVER_LIST = "servers.txt";
  public static final int SERVER_CONNECT_TIMEOUT = 1000;

  /**
   * The socket connection for listening.
   */
  private ServerSocket socket;
  /**
   * A list of all connected nodes.
   */
  private List<InetSocketAddress> servers;
  /**
   * The persistent data node.
   */
  private Node node;

  private InetSocketAddress local_address;

  private ForkJoinPool fjp;

  public Node getNode() {
    return this.node;
  }

  /**
   * Constructor specifying the port and the initial server list.
   * @param port The port.
   * @param server_list_file The initial list of all known nodes.
   */
  public NodeServer(int port, String server_list_file) {
    /* Create TCP socket */
    try {
      this.socket = new ServerSocket(port);
      this.local_address = new InetSocketAddress(this.socket.getInetAddress().getLocalHost(), this.socket.
          getLocalPort());
    } catch (IOException e) { e.printStackTrace(); System.exit(1); }
    System.out.println("==================================================");
    System.out.format("\tNode Server: %s\n", this.local_address.toString());
    System.out.println("==================================================");

    /* Connect to other node server */
    this.servers = new ArrayList<InetSocketAddress>();
    // Get list of servers
    Scanner server_list = null;
    try {
      server_list = new Scanner(new File(server_list_file));
    } catch (FileNotFoundException ex) { ex.printStackTrace(); System.exit(1); }
    // Connect to servers and store addresses
    System.out.println("Adding nodes from server file:");
    while (server_list.hasNext()) {
```

```
      String svr_host = server_list.next();
      int svr_port = server_list.nextInt();
      InetSocketAddress svr = new InetSocketAddress(svr_host, svr_port);

      if (!this.isSameAddress(this.local_address, svr)) {
        this.servers.add(svr);
        System.out.format("[+] Added node: %s\n", svr.toString());
      }
    }
    System.out.println("-------------------------------------------------");

    this.fjp = new ForkJoinPool();

    /* Initialize node */
    this.node = new Node();
    this.node.readFromFile();

    this.start();
  }

  /**
   * Processes the requests.
   */
  public void run() {
    System.out.println("Waiting for requests...");
    while (true) {
      // Accept new TCP connection
      Socket connection = null;
      try { connection = this.socket.accept(); }
      catch (IOException e) {
        e.printStackTrace();
        return; //needs to be here to avoid exceptions in the following lines
      }

      // Determine origin (client or node)
      InetSocketAddress con_addr = new InetSocketAddress(connection.getInetAddress(), connection.getPort());
      boolean from_client = true;
      for (InetSocketAddress addr : this.servers) {
        // @TODO Research if this is the best way to determine origin
        if (addr.getAddress().equals(con_addr.getAddress())) {
          from_client = false; break;
        }
      }

      new Request(connection, this, from_client);
    }
  }

  /**
   * Determines if two address are the same
   * @param addr1 The first address
   * @param addr2 The second address
   * @return True if the IP address and port are the same
   */
  public boolean isSameAddress(InetSocketAddress addr1, InetSocketAddress addr2) {
    return addr2.getAddress().equals(addr1.getAddress()) && (addr1.getPort() == addr2.getPort());
  }

  /**
   * Gets the list of all other server node addresses
   * @return The list of all other server node addresses
   */
```

```
  public List<InetSocketAddress> getServers() {
    return this.servers;
  }

  /**
   * Gets the local address of the server
   * @return The local socket address
   */
  public InetSocketAddress getLocalAddress() {
    return this.local_address;
  }

  /**
   * Gets the ForkJoinPool
   * @return The FJP
   */
  public ForkJoinPool getFJP() {
    return this.fjp;
  }

  /**
   * Starts the server.
   * @param args[0] The port to listen on.
   * @param args[1] The initial list of all other known nodes.
   */
  public static void main(String[] args) {
    int port = (args.length > 0) ? Integer.parseInt(args[0]) : DEFAULT_PORT;
    String server_list_file = (args.length > 1) ? args[1] : DEFAULT_SERVER_LIST;
    new NodeServer(port, server_list_file);
  }

  /**
   * Handles a PUT request
   * @param request The request string
   */
  public ArrayList<MusicObject> putRequest(String request) {
    Scanner sc = new Scanner(request);
    sc.next(); // throw away the PUT
    sc.useDelimiter(", ");
    String timestamp = sc.next();
    String id = sc.next();
    String artist = sc.next();
    String album_artist = sc.next();
    String album = sc.next();
    int track = sc.nextInt();
    String title = sc.next();
    int year = sc.nextInt();
    String genre = sc.next();
    int track_length = sc.nextInt();
    MusicObject mobj = new MusicObject(id, artist, album_artist, album, track,
                       title, year, genre, track_length);
    this.node.addMusicObject(mobj);
    ArrayList<MusicObject> retval = new ArrayList<MusicObject>();
    retval.add(mobj);
    return retval;
  }

  /**
   * Handles a KILL request
   * @param request The request string
   */
  public void killRequest(String request) {
```

```
    // catastropic failure... die immediately
    Runtime.getRuntime().halt(-1);
  }

}


class Request extends Thread {
  Socket socket;
  NodeServer svr;
  boolean is_leader;

  public Request(Socket connection, NodeServer svr, boolean is_leader) {
    this.socket = connection;
    this.svr = svr;
    this.is_leader = is_leader;
    this.start();
  }

  public void run() {
    System.out.format("\n[%s] %s request from: %s\n",
        (is_leader ? "C" : "N"), (is_leader ? "Client" : "Node"), this.socket.getInetAddress().toString());
    try {
      DataInputStream dis = new DataInputStream(this.socket.getInputStream());
      String request = dis.readUTF();
      System.out.println("[REQUEST] " + request);

      parseRequest(request);
    } catch (IOException e) { e.printStackTrace(); }
  }

  public void parseRequest(String request) throws IOException {
    Scanner sc = new Scanner(request);
    switch(sc.next()) {
      case "PUT":   svr.putRequest(request); break;
      case "KILL":  svr.killRequest(request); break;
      case "GET":   getRequest(request, is_leader); break;
    }
  }

  /**
   * Handles a GET request
   * @param request The request string
   * @todo This needs to be implemented.
   */
  public void getRequest(String request, boolean is_leader) throws IOException {
    if (is_leader) {
      // Distribute requests
      Collection<MusicObject> replies = this.svr.getFJP().invoke(new GetTask(request, this.svr));
      System.out.println("[%] Received replies");
      // Respond to client
      ObjectOutputStream oos = new ObjectOutputStream(this.socket.getOutputStream());
      oos.writeObject(replies);
      oos.flush();
    } else {
      // Query node
      HashMap<String, String> filter = this.svr.getNode().parseRequest(request);
      Collection<MusicObject> results = this.svr.getNode().query(filter);
      // Respond to leader node
      ObjectOutputStream oos = new ObjectOutputStream(this.socket.getOutputStream());
      oos.writeObject(results);
      oos.flush();
```

```
      System.out.format("[<] Sent node response to: %s\n", this.socket.getInetAddress().toString());
    }
  }
}

/**
 * A ForkJoinTask for GET requests
 */
class GetTask extends RecursiveTask<Collection<MusicObject>> {

  /**
   * The NodeServer
   */
  private NodeServer svr;
  /**
   * The socket connection
   */
  private Socket socket;
  /**
   * The GET request
   */
  private String request;

  /**
   * Constructor for initial request
   * @param request The GET request
   * @param svr The NodeServer
   */
  public GetTask(String request, NodeServer svr) {
    this.request = request;
    this.svr = svr;
  }

  /**
   * Constructor for each node request
   * @param request The GET request
   * @param socket The socket for the node connection
   */
  public GetTask(String request, Socket socket) {
    this.request = request;
    this.socket = socket;
  }

  /**
   * Distributes request to all connected nodes and asynchronously
   * waits for the responses
   * @return The result set for the request
   */
  @SuppressWarnings("unchecked")
  public Collection<MusicObject> compute() {
    if (this.svr != null) {
      // Create task for all node servers
      Collection<GetTask> tasks = new ArrayList<GetTask>();
      for (InetSocketAddress addr : svr.getServers()) {
        Socket s = new Socket();
        try {
          s.connect(addr, NodeServer.SERVER_CONNECT_TIMEOUT);
        } catch (ConnectException e) {
          System.err.format("[!] Connection failed: %s\n", addr.toString());
          continue;
        } catch (IOException e) { e.printStackTrace(); }
        tasks.add(new GetTask(this.request, s));
```

```
      }

      // Fork join on the replies
      Collection<Collection<MusicObject>> replies = new ArrayList<Collection<MusicObject>>();
      this.invokeAll(tasks);
      for (GetTask task : tasks) {
        replies.add(task.join());
      }

      // Add in server query
      HashMap<String, String> filter = this.svr.getNode().parseRequest(this.request);
      replies.add(this.svr.getNode().query(filter));

      // @TODO Perform consensus and return results
      for (Collection<MusicObject> result : replies) {
        //return result;
        System.out.println("[.] Reply size of: " + result.size());
      }

      ArrayList<Collection<MusicObject>> validResults = new ArrayList<Collection<MusicObject>>();
      for(Collection<MusicObject> result : replies) {
        if(result.size() != 0) validResults.add(result);
      }

      int totalNodes = validResults.size();
      int requiredNodes = totalNodes / 2 + 1;
      System.out.println("[.] INFO: totalNodes = " + totalNodes + "; requiredNodes = " + requiredNodes);

      Collection<MusicObject> master = validResults.get(0);
      validResults.remove(master);

      Collection<MusicObject> finalResults = new ArrayList<MusicObject>();

      for(MusicObject mobj : master) {
        int agree = 1;
        for(Collection<MusicObject> c : validResults)
          if(c.contains(mobj)) agree++;
        //System.out.println("[.] INFO: agree = " + agree);
        if(agree >= requiredNodes) finalResults.add(mobj);
      }

      return finalResults;

    } else if (this.socket != null) {
      try {
        // Send request to node
        DataOutputStream dos = new DataOutputStream(this.socket.getOutputStream());
        dos.writeUTF(this.request);
        dos.flush();
        System.out.format("[<] Sent node request to: %s\n", this.socket.getInetAddress().toString());

        // Wait for response
        ObjectInputStream ois = new ObjectInputStream(this.socket.getInputStream());
        Collection<MusicObject> response = (Collection<MusicObject>)ois.readObject();
        System.out.format("[>] Response received from: %s\n", this.socket.getInetAddress().toString());

        return response;
      } catch (IOException e) {
        System.out.format("[!] No response from: %s\n", this.socket.getInetAddress().toString());
        return new ArrayList<MusicObject>();
      } catch (ClassNotFoundException e) {
        System.err.println(e.getMessage());
```

```
        return new ArrayList<MusicObject>();
      }
    } else {
      throw new IllegalStateException("Invalid task.");
    }
  }
}
```

NodeServer.java