Uppsala Master's Thesis in
Computer Science 303
2005-06-14
ISSN 1100-1836

Object-Relational Wrapping of Music Files

Vaidrius Petrauskas

Information Technology
Computer Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract

Amos II (Active Mediator Object System) is a distributed mediator system that uses a functional data model and has a relationally complete functional query language, AmosQL. The purpose of this work is to use foreign Java functions and Java libraries to develop a generic wrapper facility for music files in Amos II, which extracts meta-data contained in music files of different kinds into Amos II, thus enabling searching the music files in terms of particular music files properties. To implement this, a set of primitive functions have been defined, which allow general queries to music files, and for finding all music files in a directory. A meta-data refresh facility updates extracted meta-data when music files contents have changed. A generic JavaSound API with pluggable specific music format support is used to get music meta-data from different format files. Primitive foreign functions take the name of a music file as a parameter and then store objects and attributes in the database. The system implements MP3 and Ogg Vorbis format wrappers as examples.

Supervisor: prof. Tore Risch
Examiner: prof. Tore Risch

Contents

1 Introduction

There are increasing requirements for data integration systems that combine data from different kinds of data sources and present them to users in a comprehensible format. The *wrapper-mediator* approach [21] divides the functionality of a data integration system into two kinds of subsystems. The *wrappers* (data source interfaces) provide access to the data in data. The *mediators*—data integration modules—provide coherent views of the data in the data sources by performing semantic reconciliation (handling of semantic conflicts and overlaps) between the wrapped data.

Amos II [16] is a light-weight and high-performance mediator database engine. Amos II has a functional data model with a relationally complete object-oriented query language, AmosQL. The Amos II data manager is extensible so that new data types and operators can be added to AmosQL, implemented in some external programming language (Java, C, or Lisp).

Amos II is implemented in C and runs under Windows and Linux operating systems. Amos II currently has several wrappers available, including XML [11, 17], relational databases [1, 5], RDF [15] and MIDI [10] wrappers.

Nowadays, it is very common to store music data in computers. Amos II already wraps the MIDI music format [10], which represents information on how to play music by musical instruments. One would also want to wrap musical data from other kinds of musical formats that contain the actual music data sound — like the MP3 (MPEG Audio Layer-3) file format.

The purpose of the project is to use foreign Java functions and Java libraries to develop a generic wrapper facility for music files in Amos II. With this wrapper, the Amos II system is able to search meta-data of music files and combine the retrieved data with the other wrapped Amos II data sources. Hence, Amos II technologies to query data become available for searching meta-data stored in music files, e.g., by using AmosQL to query artist and music track length information. To save storage, the system does not load the actual files themselves, but only import their meta-data. Database refresh functionality is also implemented to update imported meta-data if changed in the musical files. As an example, the wrapper is able to handle both MP3 [3] and Ogg Vorbis [22] format files. The design makes it easy to extend the wrapper with new music formats.

The generic music wrapper is defined as primitive foreign functions taking the name of a music file as a parameter and then storing Amos II objects and attributes in the database.

The primitive Java interfaces to MP3 and Ogg Vorbis files were studied to investigate how to access music data from Amos II in a generic way and transform the class structure of Java's music file API into corresponding Amos II data representations. The implemented functions allow general queries to MP3 and Vorbis files by querying extracted meta-data from the files using Java music interfaces. Some other functions are also implemented, e.g. for finding all music files in a directory.

The project was divided into the following parts:

1. Acquiring consolidated knowledge about digital music along with its representation in computing and music standards, such as MP3 and Ogg Vorbis.

2. Studying and selecting a development environment—existing generic Java multimedia specifications, frameworks, APIs and their implementations. Also, I had to choose concrete Java libraries for MP3 and Ogg Vorbis meta-data parsing.

3. Designing an object oriented database schema for the wrapper structure according to the selected Java APIs.

4. Designing primitive foreign functions to import musical meta-data given a name of a directory and to synchronize database meta-data with the real files in a file system.

5. Developing a simple playback of musical files functionality.

6. Finally, the whole application had to be integrated with the Amos II environment along with demonstration queries illustrating how to query wrapped musical files.

2 Background

In this chapter, the mediator-wrapper approach is described on which the Amos II system is based; later, the Amos II architecture itself is described. As the main purpose of Amos II is using it as a database and the music wrapper is extensively based on custom data types, the data model of Amos II is described. Finally, as this project is done with the Java programming language, Java interfaces to access Amos II system are described.

Today, a lot of applications are developed that need to access databases. Therefore, an increasing number of databases are in use. For example, online travel agencies must be able to access databases of flight companies to gain information of available seats, databases of airports to see arrival and departure timetables, and databases of hotels to search for available rooms. Such information is not stored in one database, but each flight company, airport and hotel would have their own database. Thus, access to several different databases is necessary. Furthermore, many possibilities to store data exist. One company may have one database and schema, while another company might have a completely different view on the stored data. Thus, an application needs the possibility to access heterogeneous data sources.

The mediator/wrapper approach [21] gives support for applications and users to query and access heterogeneous data sources. A mediator system consists of a mediator and one or several wrappers. A *mediator* is a virtual database that comprises a *common data model*—a data model able to specify how to map data between different data sources. The task of a mediator is to process queries sent from an application. A query is split depending on the data and capabilities of the target data sources. If, as in Amos II, a mediator may also regard other mediators as data sources, a network of mediators, consisting of several layers, can be created. In such a network only *primitive* mediators should have access to data sources. Then higher layers, consisting of more advanced data abstractions, can be queried by applications. A mediator can access different autonomous data sources, like relational databases, XML, MP3, OGG files or object stores, but never directly. Each data source needs to be accessed through software interfaces, called *wrappers*. Queries sent from the mediator are translated by a wrapper to a data source specific format and thus it hides the heterogeneity of that data source. After that, the wrapper retrieves query results that have to be translated into the common data model and passed to the mediator. There all results from all data sources are combined and returned to the application. The mediator uses data modeling primitives— e.g. types and functions to reconcile differences and overlaps of wrapped data.

2.1 Amos II

Amos II is a distributed mediator system with an object oriented and functional data model. It is based on the mediator/wrapper architecture. It is a distributed mediator system consisting of one or several mediator peers. These peers can communicate via the Internet using TCP/IP. Each peer offers its own virtual functional database layer, consisting of:

- A storage manager,
- A transaction manager,
- Data abstractions that provide transparent functional views for clients and other mediator peers to access data sources,
- A functional query language: AmosQL.

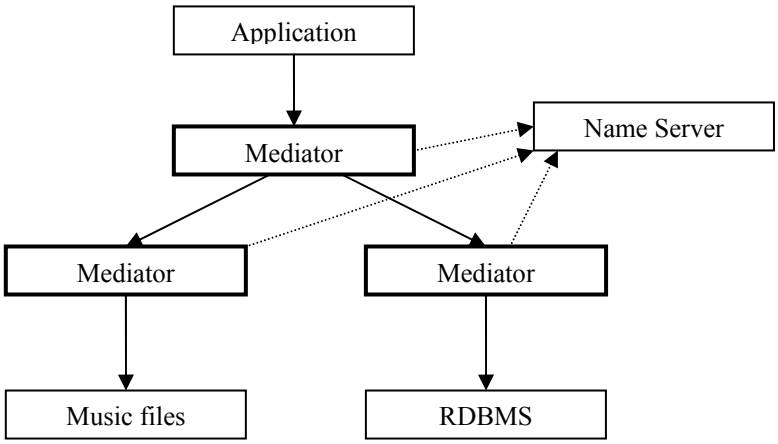

Figure 2.1.1: An example of Amos II system with three mediators

The core of Amos II is an open, light-weight and extensible database management system (DBMS). Queries to the data model are written in AmosQL [16, 6], a relationally complete functional query language. The system can be configured as a stand-alone system or as several autonomous and distributed Amos II peers. These peers can interoperate through its distributed multi-database facilities [16]. Each mediator peer offers three possibilities to access data:

- Access to data stored in an Amos II database;
- Access to wrapped data sources;

- Access to data that is reconciled from other mediator peers;

In this project a generic wrapper for music files is developed, thus making music meta-data available to mediators.

In Amos II it is possible to build up layers of peers with a dynamic communication topology. A *distributed mediator query optimizer* optimizes this communication topology. To compute an optimized execution plan for a given query, data and schema information are exchanged between the peer. In figure 2.2.1, the top mediator defines a mediating functional view integrating data from two other sub-mediators. The mediating view uses facilities for semantic reconciliation [16] for combining data from the two lower mediators. The two lower mediators translate data from a wrapped relational database and a web server, respectively. They have knowledge of how to execute AmosQL queries to get meta-data of music files, and how to translate AmosQL queries to SQL, respectively. To summarize, figure 2.2.1 gives an example of the distribution of mediator peers. Two distributed data sources offer through three distributed mediator peers their data to an application. Communication between peers is illustrated by thick lines, where the arrow indicates the peer running as server. One special mediator peer is listed in the figure, the *name server*, which stores meta-data about all peers.

Mediator peers forming a group are autonomous—there is no control of them on the name server. It is left to each peer to describe its own local data view and data sources. Is the name server involved in peer communication? No, all communication is done through messages that can request or deliver data and commands between peers. To avoid a bottleneck the name server is only involved when a new mediator peer wants to register to the group. As soon as mediators are known to each other they can communicate directly without any information from the name server. In figure 2.2 the communication with the name server is illustrated by dotted lines. The name server always acts, with regard to the other mediator peers, as a server.

2.1.1 Data Model

Basic components of the data model of Amos II are *types, functions* and *objects*. In object oriented programming languages these concepts approximately correspond to *classes, methods,* and *instances.* Types are used for classifying objects. Each object is an instance of a type. All properties of an object as well as relationships between objects are represented by functions.

2.1.1.1Types

Types correspond to entity types in Entity-Relationship-diagrams. Types unite objects with similar properties. An instance of a type is always an instance of all its supertypes. Also, if an object inherits from more than one type it gains all properties from all supertypes. Two kinds of types exist, *stored* and *derived* types. Derived types are mainly used for defining reconciled views of mediated data [16], while the stored types are defined and stored in an Amos II database. In the present work only stored types are used since the meta-data is actually imported into an Amos II database when a music file is accessed. The music contents itself stays in the file, though, to save storage and access time. The following command creates a track and an MP3 track type in Amos II:

```
create type track;
create type mp3track under track;
```

The root element in the Amos II type hierarchy is named `Object`. All (system and user) defined type definitions are stored as instances of a type named `Type`. All functions are instances of a type named `Function`. When a user defines a type it is always a subtype of type `Userobject`.

2.1.1.2Functions

Functions model the relationship between objects, properties of objects and computations over objects. In functional queries and views they are providing the basic primitives. As already mentioned, functions are instances of the type `Function`. The function's *signature* contains information about all arguments, such as the types and optional names, and about the result of the function. The next example shows the signature of a function modeling the attribute name of type `track`:

```
title(track)->charstring
```

Furthermore, functions can be *overloaded*, i.e. different functions defined for different combinations of arguments can have the same name (*polymorphism*). The selection of the correct function implementation of an overloaded function is made based on the actual argument types.

2.1.1.3Objects

Objects in Amos II correspond to entities in an ER diagram and are instances of Amos II types. Everything in Amos II is represented as an object, independent whether the object is user-defined or system-defined. *Literals* and *surrogates* are the main kinds for representing objects. Literal objects are primitive objects like integers, strings or even *collections* of other objects. In

addition to this, there are surrogates which are explicitly created by the user or the system and describe real world entities. For example, a `vorbistrack` object in Amos II can be created with a command as follows:

```
create vorbistrack instances :thetrack;
```

The system assigns unique object identifiers (OIDs) to all surrogate objects. When a query requests a surrogate object, the returned result will be displayed similar to this:

```
#[OID 1101]
```

2.1.2 External Interfaces

There are two ways to interface Amos II with other programs—either an external program calls Amos II through the *callin* interface, or Amos II calls external functions through the *callout* interface [15]. Both interfaces are used in this project.

The most convenient way to write Amos II applications is using the Java interface [4]. The music mediator is written in Java language and uses Java interfaces to communicate with Amos II.

With the *callin* interface a program in Java calls Amos II. The *callin* interface is similar in structure to the call level interfaces for relational databases, such as JDBC.

With the *callout* interface Amos II functions call methods written in Java. Each such *foreign Amos II function* is implemented by one or several Java methods. The foreign functions in Amos II are *multi-directional* which allow them to have separately defined inverses and to be indexed. The system furthermore allows the *callin* interface to be used also in foreign functions, which gives great flexibility and allows Java methods to be used as a form of *stored procedures*.

With the *callin* interface there are two ways to call Amos II from Java:

1. In the *embedded query* interface a Java method is provided that passes strings containing AmosQL statements to Amos II for dynamic evaluation. Methods are also provided to access the results of the dynamically evaluated AmosQL statement. The embedded query interface is relatively slow since the AmosQL statements have to be parsed and compiled at run time.

2. In the *fast-path* interface predefined Amos II functions are called as Java methods, without the overhead of dynamically parsing and executing AmosQL statements. The

fast-path is significantly faster than the embedded query interface. It is therefore recommended to always make Amos II derived functions and stored procedures for the various Amos II operations performed by the application and then use the fast-path interface to invoke them directly.

Amos II can be linked directly with a Java application program. This is called the *tight connection* where Amos II is an *embedded database* in the application. It provides for the fastest possible interface between an application and Amos II since both the application and Amos II run in the same address space. The disadvantages with the tight connection are that Amos II and the application must run on the same computer. Another disadvantage is that only a single Java application can be linked to Amos II.

The musical wrapper system is implemented as a set of foreign functions. The wrapper is using the *callin* interface with the tight connection to call AmosQL from inside foreign functions.

2.2 Digital Sound

What is a sound from a physical point of view? It is the result of a mechanical disturbance of some object in a physical medium, such as air. This mechanical disturbance generates vibrations that can be represented as electrical signals by means of a device (for example, a microphone), that converts these vibrations into a time-varying voltage.

An analog sound signal is the result of measuring the voltage that represents the sound. These kinds of signals are continuous in the sense that they consist of a continuum of constantly changing values. A digital sound is the result of counting all these values many times per second for a certain defined length/time. Each measurement value is called a *sample*, and this kind of process is called *sampling*. In order to process sounds on the computer, the analog sound must be converted into a digital format understandable by computer—binary numbers.

The sampling frequency of a sound is equal to the number of cycles which occur every second (*cycles per second*, abbreviated *cps* or *Hz*). The Nyquist-Shannon sampling theorem states that in order to accurately represent a sound digitally, the sampling rate must be higher than at least twice the value of the highest frequency contained in the signal [23]. The average upper limit of human hearing is approximately 18 kHz (18000 Hz), which implies a minimum sampling rate of 36 kHz (36000 Hz). The sampling rate frequently used in computer sound design systems is 44.1 kHz (44100 Hz). Currently, 48 kHz (48000 Hz) sampling rate is beginning to be used.

Another element that influences the quality of sampling is the *level of resolution*, or *quantization* of a sampler. The resolution depends upon the size of the word used to represent the amplitude of a sampling sound and is determined by the resolution of the ADC (Analog-to-Digital Converter) and DAC (Digital-to-Analog Converter). A word, for instance, could be 4-bits long or 16-bits long etc. Unsatisfactory lower resolutions are prone to cause a damaging loss of sound quality, referred to as *quantization noise*.

After digital conversion the sound needs be stored in order to be re-used or played back. The most basic way to store sound is to take the stream of samples and write them onto a file. Sound files normally contain other descriptive meta-data information related to file properties and text comments; all this information is stored in the sound file *header* (the initial portion of data of the file). In the header one may find information such as the sampling rate used, the size of the word, whether the sound is mono or stereo, and so on. There are two ways to store audio data:

1. uncompressed,
2. compressed.

Some popular uncompressed audio formats are:

- Wave, adopted by Microsoft (*.wav*)
- VOC, adopted by Creative Lab's Sound Blaster (*.voc*)
- NeXT/Sun, originated by NeXT and Sun computers (*.snd* and *.au*)
- AIFF, originated by Apple computers (*.aif*)

Uncompressed storage is often uneconomical as it might contain a great deal of redundant information like, for example, a silent portion. There are techniques for optimizing the representation of samples in order to reduce the size of the file. There are two kinds of compression:

1. Lossless.
2. Lossy.

Lossless compression is just what it sounds like—a way of compressing music into a file that, when played back, is absolutely identical to the original. Not just “sounds the same,” but that is statistically identical. The compressed, smaller file can be expanded back into the original file

without losing any information whatsoever. The original file is bit-for-bit identical. It was once only a viable option for professionals seeking to archive large volumes of audio but now large hard drives are cheap and drive-based portable players with lots of storage are abundant, so lossless formats are increasingly relevant to end users that want the best possible audio fidelity.

Some popular lossless compression formats are:

- FLAC (Free Lossless Audio Codec) by FLAC open-source project (*.flac*)
- Monkey's Audio by Matthew T. Ashland (*.ape*)
- Apple Lossless by Apple Computer, Inc. (*.m4a*)
- WMA Lossless (Windows Media Audio) by Microsoft (*.wma*)

Lossy compression removes some information in order to make the file easier to compress. Lossy compression is by far the most popular format, because it allows for much smaller file sizes. Virtually all lossy compression schemes, whether for audio or video, work by a principle called *perceptual coding*. This is the process of removing parts of the original data that the user will probably not perceive anyway. The trick is to remove as little information as possible from the original audio sample and to make sure that what is removed is hard to hear (frequencies above the range of human hearing or computer equipment capabilities, for example). Lossy audio compression formats such as Ogg Vorbis and MP3 can achieve results which are provably indistinguishable from the original, CD-quality, sound but are a mere 10 to 20% of the size. Lossy compression formats widely used today are:

- MP3 (MPEG Audio Layer 3) by Fraunhofer Institut für Integrierte Schaltungen (*.mp3*)
- Ogg Vorbis by Xiph.org (*.ogg*)
- WMA (Windows Media Audio) by Microsoft (*.wma*)
- AAC (MPEG-4 Advanced Audio Coding) by Dolby, Fraunhofer, AT&T, Sony, and Nokia (*.aac*, *.m4a*).

Sound artists and composers normally do not work with compressed sounds. Compression should be applied only after the piece is completed.

2.2.1 MP3

The MPEG acronym stands for *Motion Picture Experts Group* and it refers to a group of researchers who study new formats for coding and playing audio and video; the acronym refers to audio/video compression formats created by this group. The term MP3 is the abbreviation of MPEG1-Layer 3, which is the audio compression format, used in the MPEG 1 algorithm [3]. In other words, MPEG is a series of compression algorithms to reproduce audio and video; the layers are compression algorithms used in MPEG playing only for audio; MPEG 1-Layer 3, known as MP3, is one of the audio compression algorithm used by MPEG 1 algorithm.

MPEG-1 is the name for the first phase of MPEG work, starting in 1988 [2]. This work was finalized with the adoption of the ISO/IEC standard in late 1992. The audio coding part of this standard describes a generic coding system, designed to fit the demands of many applications. MPEG-1 Audio consists of three operating modes called *layers*, with increasing complexity and performance, named Layer-1, Layer-2 and Layer-3. Layer-3, with the highest complexity, was designed to provide the highest sound quality at low bit-rates (around 128 kbit/s for a typical stereo signal).

MPEG-2 denotes the second phase of MPEG. It introduced a lot of new concepts into MPEG video coding, including support for interlaced video signals. The main application area for MPEG-2 is digital television. The original MPEG-2 Audio standard was finalized in 1994 and consisted of two extensions to MPEG-1 Audio:

- Multi-channel audio coding, including the 5.1 channel configuration well known from cinema sound – this multi-channel extension is done in a backward compatible way, allowing MPEG-1 stereo decoders to reproduce a mixture of all available channels.

- Coding at lower sampling frequencies – this extension adds sampling frequencies of 16 kHz, 22.05 kHz and 24 kHz to the MPEG-1 sampling frequencies of 32 kHz, 44.1 kHz and 48 kHz, improving the coding efficiency at very low bit-rates.

MPEG-1 Audio works for both mono and stereo signals. A technique called joint stereo coding can be used to achieve a more efficient combined coding of the left and right channels of a stereophonic audio signal. Layer-3 allows both mid/side stereo coding and intensity stereo coding. The latter is especially helpful for lower bit-rates, but bears the risk of changing the sound image. The operating modes are:

1. single channel;
2. dual channel (two independent channels, for example containing different language versions of the audio);
3. stereo (no joint stereo coding);
4. joint stereo.

MPEG audio compression works on a number of different sampling frequencies. MPEG-1 defines audio compression at 32 kHz, 44.1 kHz and 48 kHz. MPEG-2 extends this to half the rates, i.e. 16 kHz, 22.05 and 24 kHz. "MPEG-2.5" is the name of a proprietary extension to Layer-3, developed by Fraunhofer IIS, which introduces the sampling frequencies 8 kHz, 11.05 kHz and 12 kHz.

MPEG Audio does not just work at a fixed compression ratio. The selection of the bit rate of the compressed audio is, within some limits, completely left to the implementer or operator of an MPEG audio coder. For Layer-3, the standard defines a range of bit-rates from 8 kbit/s up to 320 kbit/s. Bit-rate is proportional to sampling frequency. Furthermore, Layer-3 decoders must support the switching of bit-rates from audio frame to audio frame. Combined with the bit reservoir technology, this allows both variable bit-rate coding and constant bit-rate coding at any fixed value within the limits set by the standard.

A very important property of the MPEG standards is the principle of minimizing the amount of normative elements in the standard. In the case of MPEG Audio, this led to the fact that only the data representation, i.e. the format of the compressed audio, and the decoder are normative. Even the decoder is not specified in a bit-exact fashion. Instead, formulae are given for most parts of the algorithm, and compliance is defined by a maximum deviation of the decoded signal from a reference decoder, implementing the formulae with double-precision arithmetic accuracy. This allows building decoders running both on floating-point and fixed-point architectures.

2.2.2 Ogg Vorbis

Ogg Vorbis is a fully open, non-proprietary, patent-and-royalty-free, and general-purpose compressed audio format for mid to high quality (8 kHz-48.0 kHz, 16+ bit, polyphonic) audio and music at fixed and variable bit rates from 16 to 128 kbps/channel. This places Vorbis in the same

competitive class as audio representations such as MPEG-4 (AAC), and similar to, but higher performance than MPEG-1/2 audio layer 3, MPEG-4 audio (TwinVQ), WMA and PAC.

Technically, Ogg is a patent-free, fully open multimedia bitstream container designed for efficient streaming and storage. It is often used incorrectly to refer to the audio codec Ogg Vorbis. It has been created as the framework of a larger initiative aimed at developing a set of components for the coding and decoding of multimedia content which are both freely available and freely re-implementable in software. The Ogg transport bit stream is designed to provide framing, error protection and seeking structure for higher-level codec streams that consist of raw, unencapsulated data packets, such as the Vorbis audio codec or Theora video codec.

The format consists of chunks of data each called an Ogg Page [22]. Each page begins with the "OggS" string which can be used to identify the file as Ogg. A serial number and page number in the page header identifies each page as part of a series of pages which make up a bit stream. Multiple bitstreams may be multiplexed (combined into one stream) in the file where pages from each bit stream ordered by the seek time of the contained data. Bit streams may also be appended to existing files, a process known as *chaining*, to cause the bit streams to be decoded in sequence.

Currently, it is possible to embed these bit streams:

- Audio codecs:
    - Lossy:
        - Speex (voice data at low bitrates)
        - Vorbis (general audio data at mid- to high-level bitrates)
    - Lossless:
        - FLAC (archival and high-fidelity audio data)
- Text codec:
    - Writ (a text codec designed to embed subtitles and captions)
- Video codecs:
    - Theora

- Tarkin

Ogg Vorbis is the flagship open source multimedia project of Xiph.Org Foundation. It was launched after the copyright holders of the MP3 codec (coder/decoder) closed the source of their product and began demanding stringent licensing fees. In late 1998, Fraunhofer IIS, who owns the rights to the MP3 codec, sought royalties for all implementations of their MPEG Layer 3 audio codec. Around this time, MP3 was a mainstream audio format and used by millions of users worldwide.

Given 44.1 kHz (standard CD audio sample frequency) stereo input, the current encoder as of September 2004 will produce output from 45 to 500 kbit/s depending on the specified quality setting. Quality settings run from -1 to 10 and are an arbitrary metric; files encoded at -q5, for example, should have the same quality of sound in all versions of the encoder, but newer versions should be able to achieve that quality with a lower bit rate. Vorbis is inherently variable bit rate (VBR).

Vorbis uses the modified discrete cosine transform (MDCT) for converting sound data from the time domain to the frequency domain. The resulting frequency-domain data is broken into noise floor and residue components, and then quantized and entropy coded using a codebook-based vector quantization algorithm. The decompression algorithm reverses these stages. The noise floor approach gives Vorbis its characteristic analog noise -like failure mode (when the bit rate is too low to encode the audio without perceptible loss), which many people find more pleasing to the ears than metallic warbling as in MP3.

Many users feel that Vorbis reaches transparency (sound quality that is indistinguishable from the original source recording) at a quality setting of -q5, approximately 160 kbit/s. For comparison, it is commonly felt that MP3 reaches transparency at around 192 kbit/s, resulting in larger file sizes for the same sound quality.

2.2.3 Java Sound APIs

There are two existing Java APIs for working with sound:

1. Java Sound

2. Java Multimedia Framework (JMF)

The Java Sound API [18] is a low-level API for effecting and controlling the input and output of sound media, including both audio and Musical Instrument Digital Interface (MIDI) data [7]. The Java Sound API provides explicit control over the capabilities normally required for sound input and output, in a framework that promotes extensibility and flexibility.

The Java Sound API provides the lowest level of sound support on the Java platform. It provides application programs with a great amount of control over sound operations, and it is extensible. For example, the Java Sound API supplies mechanisms for installing, accessing, and manipulating system resources such as audio mixers, MIDI synthesizers, other audio or MIDI devices, file readers and writers, and sound format converters. The Java Sound API does not include sophisticated sound editors or graphical tools, but it provides capabilities upon which such programs can be built. It emphasizes low-level control beyond that commonly expected by the end user.

The Java Media Framework (JMF) [7] is a higher-level API that is currently available as a Standard Extension to the Java platform. JMF specifies a unified architecture, messaging protocol, and programming interface for capturing and playing back time-based media. JMF provides a simpler solution for basic media-player application programs, and it enables synchronization between different media types, such as audio and video. On the other hand, programs that focus on sound can benefit from the Java Sound API, especially if they require more advanced features, such as the ability to carefully control buffered audio playback or directly manipulate a MIDI synthesizer.

For this project, Java Sound API is sufficient, so below I describe Java Sound API.

The Java Sound API includes support for both digital audio and MIDI data. These two major modules of functionality are provided in separate packages:

- `javax.sound.sampled` (for capture, mixing, and playback of digital (sampled) audio)
- `javax.sound.midi` (for MIDI synthesis, sequencing, and event transport)

Two other packages permit service providers (as opposed to application developers) to create custom components that can be installed on the system:

- `javax.sound.sampled.spi`
- `javax.sound.midi.spi`

This project deals with sampled audio only, so let's discuss the sampled audio system. The `javax.sound.sampled` package handles digital audio data, also referred to as *sampled audio*.

Java Sound does not assume a specific audio hardware configuration; it is designed to allow different kinds of audio components to be installed on a system and accessed by the API. Java Sound supports common functionality such as input and output from a sound card (for example, for recording and playback of sound files) as well as mixing of multiple streams of audio.

Java Sound doesn't directly support MP3 or Ogg Vorbis file formats. Support for these formats is available through third-party libraries with use SPI—Service Provider Interface [19].

The main concepts of Java Sound are:

- *Lines:* A line is an element of the digital audio "pipeline," such as an audio input or output port, a mixer, or an audio data path into or out of a mixer. The audio data flowing through a line can be mono or multi-channel (for example, stereo).

- *Controls*: Data lines and ports often have a set of controls that affect the audio signal passing through the line. The way in which the signal is affected depends on the type of control. Examples are: `GainControl` (allows the signal's volume to be boosted or cut a specified number of decibels), `PanControl` (affects the sound's right-left positioning).

- *Audio System:* The `AudioSystem` class serves as an application's entry point for accessing the installed sampled-audio resources. `AudioSystem` can be queried to learn what kinds of audio components have been installed, and then one can obtain access to them, for example, mixers, lines, format conversions, files, and streams.

- *System Configuration (SPI classes)*: Service provider interfaces for the sampled audio system are defined in the `javax.sound.sampled.spi` package. Service providers can extend the classes defined here so that their own audio devices, sound file parsers and writers, and format converters can be installed and made available by a Java Sound implementation.

2.2.4 MP3 and Ogg Vorbis SPI

MP3 SPI for Java Sound and Ogg Vorbis SPI for Java Sound are two open-source projects from JavaZOOM [8].

- MP3SPI [12] is a Java Service Provider Interface that adds MP3 (MPEG 1/2/2.5 Layer 1/2/3) audio format support for the Java Platform. It supports streaming, ID3v2 frames, equalizer etc. It is based on JLayer and Tritonus Java libraries [20].

- VorbisSPI [14] is a Java Service Provider Interface that adds OGG Vorbis audio format support to Java platform. It supports Icecast streaming. It is based on JOrbis [14] and Tritonus Java libraries.

Both projects use Tritonus [20]—an open-source implementation of the Java Sound API. Originally, it emerged as a Linux-only Java Sound API implementation with support for various Linux sound systems, but now it is being used on Windows systems as well.

Only Java 5 Standard Edition 1.5 allows to pass audio properties as a map (`java.util.Map` class). However, MP3 SPI and Ogg Vorbis SPI provide a workaround to get these audio properties for J2SE 1.3 and J2SE 1.4—class cast from `AudioFileFormat` to `TAudioFileFormat` provided by Tritonus library. This way we get the same properties mechanism (map of key: value pairs) in J2SE 1.3, 1.4 and Java 1.5.

3 System Architecture

The project includes a database schema to describe musical data. When using the wrapper, the user has to extract meta-data of music files into the database populated according to the schema: the wrapper creates objects and sets their properties. Then, the user can make queries to the database using these stored objects and/or their properties. Later, if database information gets out of sync with the music files in the file system, the user can refresh the database information. During refresh, all directories known to the wrapper are scanned recursively for added/deleted directories and changed music files.

To differentiate between music and other files, the wrapper implements file extension *hints*, indicating if a file extension stores a musical format known by the wrapper. The actual format stored in a music file is determined automatically by the JavaSound API itself. Later, when wrapper needs to know the actual format of an imported music file so it knows what objects of what type it should create, the wrapper checks a meta-data property which always must be present for a given format, but which does not exist for any other format. File extension hints are implemented because it would be slower for JavaSound API to try to load every file, parse it and fail, so instead it tries to load at least the files with the right file extension that the wrapper implements.

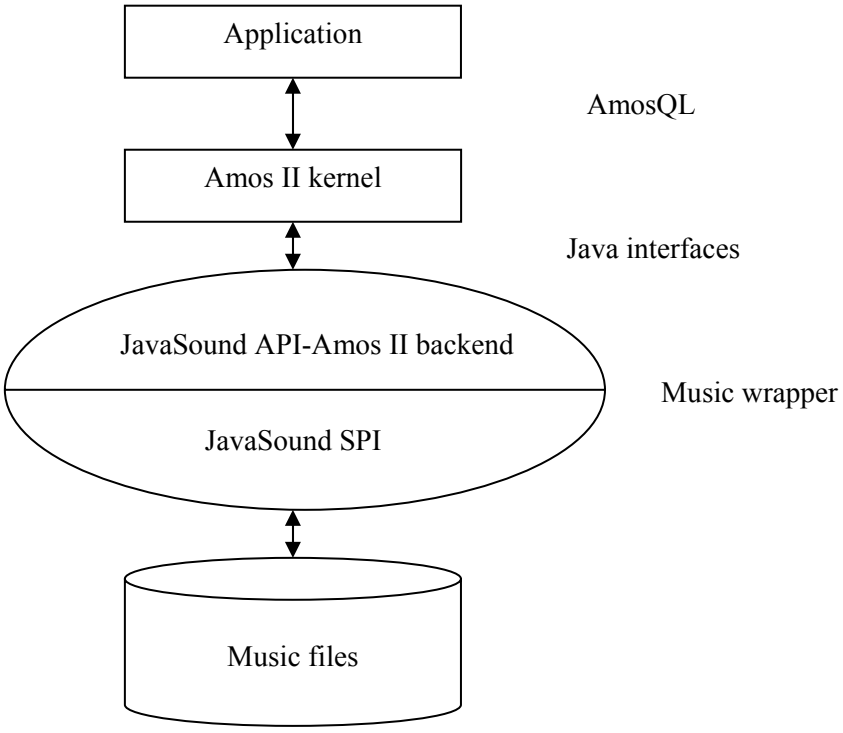

Figure 3.1. Wrapper architecture

3.1 Entity-Relationship-Attribute Schema

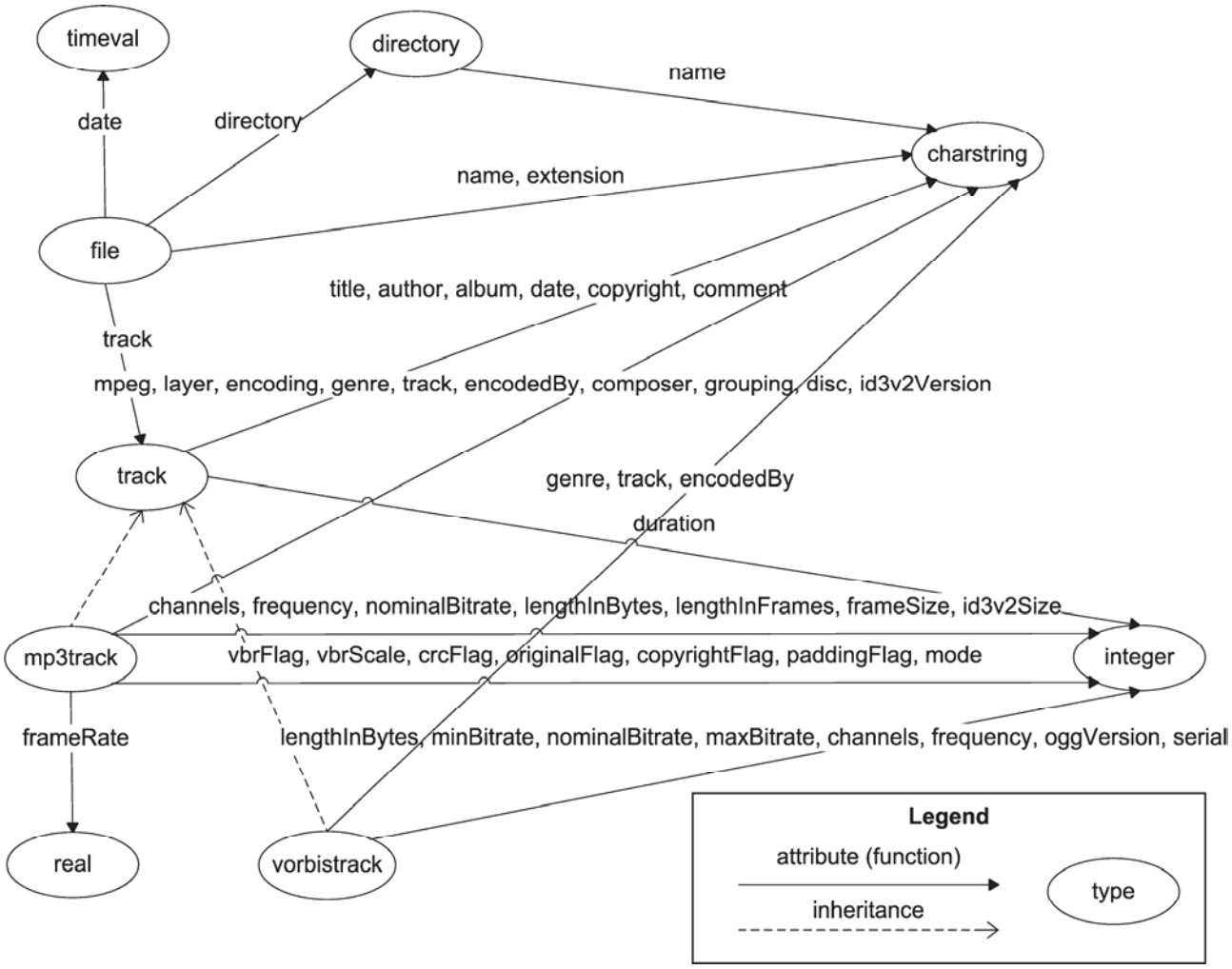

Figure 3.1.1. Entity-Relationship-Attribute schema.

Figure 3.1.1 depicts the database schema as an extended entity-relationship diagram. The diagram includes not only attributes represented as stored Amos II functions, but also derived functions.

The schema consists of these basic types and their attributes:

`directory` type properties:

Name	Amos type	Description
`Name`	`Charstring`	absolute directory path

Table 3.1.1: **`directory`** type properties

`file` type properties:

Name	Amos type	Description
`Name`	`Charstring`	absolute file name
`extension`	`Charstring`	file extension
`Date`	`Timeval`	file last modified date
`directory`	`Directory`	a corresponding directory object
`Track`	`Track`	a corresponding track object

Table 3.1.2: **`file`** type properties

Type `track` has one attribute—the property list—which is a stored function that takes track and property name as parameters and returns a `literal` value. To make properties user-friendly, derived functions are defined which select corresponding property from the property list. The following derived attributes accessing such properties are defined for all audio tracks:

`track` type properties:

Name	Amos type	Java name	Java type	Description
`duration`	`integer`	`"duration"`	`Long`	duration in microseconds
`Title`	`charstring`	`"title"`	`String`	title of the track
`Artist`	`charstring`	`"author"`	`String`	name of the artist of the track
`Album`	`charstring`	`"album"`	`String`	name of the album of the track
`Date`	`charstring`	`"date"`	`String`	the date (free-form string) of the recording or release of the track
`copyright`	`charstring`	`"copyright"`	`String`	copyright message
`Comment`	`charstring`	`"comment"`	`String`	comment of the track

Table 3.1.3: **`track`** type properties

The properties above belong to the Java class `AudioFileFormat` (i.e. not the specific SPI classes). Enclosed in quotes is the property key in the property list. The same property key is used in a Java SPI property map. Some properties might not exist for a given file.

3.2 MP3 Schema

The MP3 database schemas contains only one type—`mp3track` which is inherited from `track`. `mp3track` itself has no stored attributes. As with `track` type, there is only property list and derived functions accessing properties special for MP3 data.

`mp3track` type properties:

Name	Amos type	Java name	Java type	Description
`Mpeg`	`charstring`	"mp3.version.mpeg"	`String`	MPEG version: 1, 2 or 2.5
`Layer`	`charstring`	"mp3.version.layer"	`String`	MPEG layer version: 1, 2 or 3
`Encoding`	`charstring`	"mp3.version.encoding"	`String`	MPEG encoding: MPEG1, MPEG2-LSF, MPEG2.5-LSF
`Channels`	`Integer`	"mp3.channels"	`Integer`	number of channels: 1 (mono), 2 (stereo)
`Frequency`	`Integer`	"mp3.frequency.hz"	`Integer`	sampling rate in Hz
`nominalBitrate`	`Integer`	"mp3.bitrate.nominal.bps"	`Integer`	nominal bitrate in bits per second
`lengthInBytes`	`Integer`	"mp3.length.bytes"	`Integer`	length in bytes
`lengthInFrames`	`Integer`	"mp3.length.frames"	`Integer`	length in frames
`frameSize`	`Integer`	"mp3.framesize.bytes"	`Integer`	frame size of the first frame. It is not constant for VBR

				tracks
frameRate	Real	"mp3.framerate.fps"	Float	frame rate in frames per second
id3v2Size	Integer	"mp3.header.pos"	Integer	position of first audio header (or ID3v2 size)
vbrFlag	Integer	"mp3.vbr"	Boolean	variable bitrate flag
vbrScale	Integer	"mp3.vbr.scale"	Integer	variable bitrate scale
crcFlag	Integer	"mp3.crc"	Boolean	cyclical redundancy check flag
originalFlag	Integer	"mp3.original"	Boolean	original flag
copyrightFlag	Integer	"mp3.copyright"	Boolean	copyright flag
paddingFlag	Integer	"mp3.padding"	Boolean	padding flag
Mode	Integer	"mp3.mode"	Integer	mode: 0: STEREO 1: JOINT_STEREO 2: DUAL_CHANNEL 3: SINGLE_CHANNEL
Genre	charstring	"mp3.id3tag.genre"	String	ID3 tag genre
Track	charstring	"mp3.id3tag.track"	String	ID3 tag track info
encodedBy	charstring	"mp3.id3tag.encoded"	String	ID3 tag encoded by info
Composer	charstring	"mp3.id3tag.composer"	String	ID3 tag composer info
Grouping	charstring	"mp3.id3tag.grouping"	String	ID3 tag grouping info

Disc	charstring	"mp3.id3tag.disc"	String	ID3 tag disc info
id3v2Version	charstring	"mp3.id3tag.v2.version"	String	ID3v2 tag major version (2=v2.2.0, 3=v2.3.0, 4=v2.4.0)

Table 3.2.1: **mp3track** type properties

The only property left unused from those offered by MP3 SPI is "mp3.id3tag.v2", which returns a Java InputStream of ID3v2 frames. Also, this wrapper supports local file system files only, so properties for Shoutcast (a technology for broadcasting MP3 files on the Internet in real-time) streaming are not extracted.

3.3 Ogg Vorbis Schema

Ogg Vorbis database schema also contains only one type—vorbistrack which is inherited from track. vorbistrack itself has no stored attributes. As with track and mp3track types, there is only a property list and derived attributes.

vorbistrack type properties:

Name	Amos type	Java name	Java type	Description
lengthInBytes	Integer	"ogg.length.bytes"	Integer	length in bytes
minBitrate	Integer	"ogg.bitrate.min.bps"	Integer	minimum bitrate
nominalBitrate	Integer	"ogg.bitrate.nominal.bps"	Integer	nominal bitrate
maxBitrate	Integer	"ogg.bitrate.max.bps"	Integer	maximum bitrate
channels	Integer	"ogg.channels"	Integer	number of channels: 1 (mono), 2 (stereo)
frequency	Integer	"ogg.frequency.hz"	Integer	sampling rate in Hz
oggVersion	Integer	"ogg.version"	Integer	version of Ogg standard

Serial	Integer	"ogg.serial"	Integer	serial number of track
Genre	charstring	"ogg.comment.track"	String	genre of track
Track	charstring	"ogg.comment.genre"	String	track number of track
encodedBy	charstring	"ogg.comment.encodedby"	String	encoder name, used to encode the file

Table 3.3.1: **vorbistrack** type properties

Properties left unused are extended indexed comments with custom names. Some properties might not exist for a given file.

3.4 System Usage

The system is started by invoking the Amos II system with the wrapper's database image. Then, if the function `load` is executed to import the meta-data, it takes one string parameter, specifying a file or a directory path. If a directory is specified the system recursive accesses all files in the directory and its subdirectories. For each accessed file, its meta-data is imported, the corresponding database objects are created, and their properties are populated. For example, the following command loads meta-data of all files in "test" directory to Amos II. Then user can make queries for these objects and their properties:

```
load("test");
```

If the user wants to get name of artist, name of song, and a duration of all songs by The Erm, the query for this is:

```
select name(t), duration(t) from track t where artist(t)="The Erm";
```

The following query selects all MP3 files which have Original bit set:

```
select title(t) from mp3track t where originalFlag(t)=1;
```

The following query returns a list of all MP3 files whose extension is not "mp3":

```
select name(file(t)) from mp3track t where not(like_i(extension(file(t)), "mp3"));
```

A nice feature is to start playing accessed songs simultaneously. Playing a track opens a new graphical window.

The imported meta-data may get out-of-synch from the musical files if new musical files have been added or if old MP3 files have been replaced or updated. The system remembers last modified date of an imported file and checks it against the real one, this way determining if file needs updating. The file list is built recursively and checked against all files in a database, thus checking which files' meta-data needs to be extracted or removed. Hence the user can resynchronize (refresh) information in the database with the updated data in the file system by calling the function `update()` with no parameters. It scans all known directory paths recursively for added/deleted files and directories and adds/deletes corresponding objects and their properties to Amos II:

```
update();
```

User can also save Amos database image using command:

```
save "image.dmp";
```

Amos II can later be started with this image (file name passed as a parameter), and the user will be able to continue using accumulated meta-data in the database. The user then also can refresh the database.

4 Implementation

The wrapper is implemented using AmosQL code and four Java foreign functions. The Java functions implement only the parts of the wrapper that could not be implemented in AmosQL.

The meta-data loader, which imports meta-data from the music files into the mediator's database, is implemented with the function `load` that takes parameter of file name or directory name. If a directory name is specified, the foreign Java function `getFiles` builds a recursive list of all files in that directory, using a file extension filter (AmosQL function `fileExtHints`). `getFiles` function also has a side-effect of creating directory objects in database, so that these directories remain known to the database and can be later refreshed. Then, each file name one by one is passed to `loadFile` foreign Amos function, which does the actual parsing of the file and stores meta-data in the database.

The Java function `load` both imports new meta-data to the database and refreshes existing file meta-data. This is how it works:

1. It retrieves the property list of the properties of the music file. Firstly, it gets a file name of the music file by parsing the Amos II command line parameter, and then builds a chain of Java Sound API classes to get a property map of all available properties for this particular file and their values.

2. The derived function `trackstests` is called. For every result—a bag of track type and a test for this type, the function checks if the test property is available for current file. If it is, then conclusion is made that this is the right type of track (e.g., `mp3.id3tag.v2.version` determines the type `mp3track`, which always exists for MP3 files and not for any other music format).

3. The stored procedure `getFile` is called with arguments of file name, file last modified year, month, day, hour, minute, and second by using corresponding functions from Java API. This procedure searches the database for an already existing file object with the file name. If it exists, the procedure returns it, otherwise, creates a new one, sets a file name attribute, and returns it. Also, if the file object exists, the procedure compares the last modified date of the one stored in database and the one given as parameter. If they are not the same, this means that file object needs updating, or, if the procedure created a new object, it also means that properties must be set. Having determined this, the procedure returns a set of

file objects, and integer flags indicating whether the `load` function should be continuing its work to update file properties (it should if file last modified date is different from the one in database).

4. If function `load` needs to update some `file` properties. It sets attribute `extension` of the `file`, gets a corresponding directory object (and returns it at the end of execution) using the `getDirectory` stored procedure and then sets a `directory` attribute of `file` object.

5. Then function `load` checks if stored function `track` is set for `file` object. If it is, it means that we are updating a file (attribute was set before). If not, a new object of type which was determined in step 2 is created, and set to `track` attribute of `file`.

6. Then a `type` object of type "`track`" is retrieved from the database using `getTrackType` derived function. The function gets a set of all Java property names (and their Java types); this set is defined by the type of track. For example, one would not need Vorbis-specific properties for an MP3 track. These property names will be used later to see what properties should the function query and emit to database.

7. For every property function the system checks if this function exists, if it does, the system sets a property in the database (casting value to proper type first). If the function does not exist in the current file, but it exists in the database, the system removes it from database.

The database meta-data information is refreshed in a similar way; foreign function `update` with no parameters is used. It does this:

1. All existing file names are retrieved from the database using `oldFileNames` derived function. This function selects names of all `file` objects.

2. All file names from file system are retrieved using `newFileNames` derived function. This function gets them in the same way as `load` gets existing file names (by using `fileExtHints` and `getFiles` foreign functions).

3. For every "old" file name, it checks if the file exists. If not, it deletes file meta-data from the database using the `deleteFile` stored procedure, else, it passes file name to `load` Java function to refresh it.

4. The refresh function updates directory information saved in the database. For every directory name (retrieved using `dirNames` derived function), it checks whether the directory exists. If not, it removes its object using `deleteDirectory` stored procedure.

5. Finally, for every file which is not among old files and is among new ones, the Java function `load` is called to store its meta-data in the database.

A separate Java class `PlayThread` is implemented for playing music files with a simple GUI. There is an AmosQL function `play` which takes a track as a parameter. The actual foreign function implementing this is `playFile`, which takes file name as a parameter. It creates `PlayThread` class, sets its properties and then executes it in a new thread. `PlayThread` does the following:

1. It creates a minimal GUI, containing playing song file name and a button "Stop".

2. It constructs `AudioInputStream` from `File`, which is constructed using given file name.

3. It constructs `AudioInputStream` which will be decoded by underlying SPI by converting from `AudioFormat` which file is in to a decoded `AudioFormat`.

4. It gets `SourceDataLine` from `AudioSystem` for decoded format, opens it, feeds audio data to it with 4096 byte blocks. If the user waits to finish playing (drains line) it stops.

4.1 Limitations

In J2SE 1.4 and earlier, a way to get properties is inflexible, static and allows only few common predefined properties. Since Java 5, a new type of passing properties is allowed—by immutable `java.util.Map` key/value pairs.

As a compromise to support Java 1.4 and earlier and still have advanced properties this project uses alternative JavaSound implementation—open-source *Tritonus* library, which uses the same mechanism but with class-cast workaround. SPI implementations often provide separate versions for Java 1.5 and for Tritonus. MP3SPI, Ogg Vorbis SPI and many others have versions for Tritonus implementation, and only recently for Java 1.5.

4.2 Adding new format

An important aim of this project is extensibility—the wrapper should be designed so that it would be easy to add a new music format. The wrapper is initially implemented for MP3 and Ogg

Vorbis formats. We will take an example of JMAC SPI library which implements support for Monkey's Audio (`.ape`) [13, 9] format:

1. Make Java SPI libraries for this new format available to Java runtime. This can be done copying them to "`lib\`" directory and specifying them in "`env.cmd`" file. In JMAC case, add "`jmactritonusspi.jar`" to Java classpath in "`env.cmd`" file

2. Create new AmosQL file and add a command to parse it to "`mkSchema.amosql`" file, e. g. mkApeSchema. In this new file, one needs to:

   a. Specify create new track type under the type `track`, e. g.

   ```
   create type apetrack under track;
   ```

   b. Specify a test property form Java property list, which always exist for a given format (and for no other one), usually format/encoder version. The wrapper uses this property to check the musical format of current file, so it would know the type of track it should create (e.g., `mp3track` vs. `apetrack`):

   ```
   set tracktest(typenamed("APETRACK")) = "ape.version";
   ```

   c. Specify all properties for the format and their Java types. One example:

   ```
   add props(typenamed("APETRACK")) = <"ape.peaklevel", "Integer">;
   ```

   d. Specify derived helper functions for more comfortable use. One example:

   ```
   create function peakLevel(apetrack tr) -> integer as select
   property(tr, "ape.peaklevel");
   ```

3. Edit file extension hints function in "`mkBasicSchema.amosql`" file to include "`.ape`" files as well. For example,

   ```
   create function fileExtHints(charstring fn) -> boolean as
   select where like_i(fn, "*.mp3") or
                like_i(fn, "*.ogg") or
                like_i(fn, "*.ape");
   ```

4.3 Performance

Tests were performed in two real-life collections of MP3 and Ogg files. In the first test, the meta-data of 820 files collection was loaded into database and loading took 57 seconds (14.4

files/sec). In second test, meta-data of a collection of 3924 files was loaded into database, and that took 246 seconds (15.9 file/sec). In general, performance is acceptable.

5 Summary

Nowadays, information systems are increasingly distributed by means of computer networks and the Internet. Information flows are very dynamic and continuously changing. The Amos II mediator database implements an approach to this problem by dividing functionality of data integration to two subsystems—wrappers, providing access to data in various data sources using common data model, and mediators—servers with one or more wrappers, providing coherent views of data from wrappers.

The purpose of this project was to extend Amos II with a generalized music file wrapper to wrap different kinds of music files. Today, music files make up a large proportion of data available, so there is a need to have a way to query different kinds of music files. This wrapper enables Amos II to query music files of various formats. As an example, MP3 and Ogg Vorbis format wrappers were implemented. This wrapper does not deal with reconciliation between data from different sources. In future, such reconciliation could be designed and implemented.

The wrapper makes a tradeoff for compatibility with Java 1.4 APIs—using an external Tritonus JavaSound API implementation which is similar to Java 5 one. This could be improved by, for example, adding dynamic support for both Java 5 and Tritonus implementations. Also, editing of meta-data in database and saving it back to files could be implemented. There could also be query functions for audio fragments comparison and search.

6 References

[1]. S. Brandani: *Multi-database Access from Amos II using ODBC.* In Linköping Electronic Press, Vol. 3, Nr. 19, Dec. 8th, 1998. *http://www.ep.liu.se/ea/cis/1998/019/*

[2]. K. Brandenburg and H. Popp: *An introduction to MPEG Layer-3*. EBU Technical Review, Fraunhofer Institut für Integrierte Schaltungen (IIS), Germany, June 2000. *http://www.mp3-tech.org/programmer/docs/trev_283-popp.pdf*

[3]. P. Corsini: *The MP3 standard*. April 2, 1999. *http://www.hwupgrade.com/audio/diamond_rio/index2.html*

[4]. D. Elin, T. Risch: *Amos II JAVA Interfaces.* UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden, 2000. *http://user.it.uu.se/~torer/publ/javaapi.pdf*

[5]. G. Fahl and T. Risch: *Query Processing over Object Views of Relational Data*. The VLDB Journal, Vol. 6 No. 4, November 1997, pp 261-281.

[6]. S. Flodin, M. Hansson, V. Josifovski, T. Katchaounov, T. Risch, and M. Sköld: *Amos II Release 7 User's Manual*, March 29, 2005. *http://www.dis.uu.se/~udbl/amos/doc/amos_users_guide.html*

[7]. *Java Plug-in Developer Guide.* June 11, 2005. *http://java.sun.com/j2se/1.4.2/docs/guide/sound/programmer_guide/contents.html*

[8]. JavaZOOM team. http://www.javazoom.net

[9]. *JMAC: Open Source Java Monkey's Audio Decoder/Encoder/JavaSound SPI. http://jmac.sourceforge.net/*

[10]. M. Jost: *A Wrapper for MIDI files from an Object-Relational Mediator System*, Technical Report, Uppsala Database Laboratory. *http://user.it.uu.se/~udbl/publ/MidiWrapper.pdf*

[11]. H. Lin, T. Risch, and T. Katchanounov: *Adaptive data mediation over XML data.* In special issue on "Web Information Systems Applications" of Journal of Applied System Studies (JASS), Cambridge International Science Publishing, 2002.

[12]. *MP3 SPI for Java Sound. http://www.javazoom.net/mp3spi/mp3spi.html*

[13]. *Monkey's Audio. http://www.monkeysaudio.com/*

[14]. *Ogg Vorbis SPI for Java Sound. http://www.javazoom.net/vorbisspi/vorbisspi.html*

[15]. T. Risch: *Functional Queries to Wrapped Educational Semantic Web Meta-data* in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, Springer, ISBN 3-540-00375-4, 2003.

[16]. T. Risch, V. Josifovski, and T. Katchaounov: *Functional Data Integration in a Distributed Mediator System*, in P. Gray, L. Kerschberg, P. King, and A. Poulovassilis (eds.): *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, Springer, ISBN 3-540-00375-4, 2003.

[17]. C. Rodunger: *Accessing XML data from an Object-Relational Mediator Database*. Uppsala Master's Theses in Computing Science no. 235, ISSN 1100-1836, 2002.

[18]. Sun Microsystems, Inc.: *Java Sound API*, January 11, 2002, *http://java.sun.com/j2se/1.4.2/docs/guide/sound/programmer_guide/contents.html*

[19]. Sun Microsystems, Inc.: *The Science of Java Sound*, June 1999, *http://java.sun.com/developer/technicalArticles/Media/JavaSoundAPI/*

[20]. *Tritonus: Open Source Java Sound*. *http://www.tritonus.org/*

[21]. G. Wiederhold: Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 1992.

[22]. Wikipedia: *Ogg*, June 10, 2005, *http://en.wikipedia.org/wiki/Ogg*

[23]. Wikipedia: *Nyquist-Shannon sampling theorem*, June 06, 2005. *http://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem*