CHALMERS

Metadex

A low-latency multimedia indexer for In-Vehicle Infotainment

JONATAN PÅLSSON

NICLAS TALL

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Metadex
A low-latency multimedia indexer for In-Vehicle Infotainment

Jonatan Pålsson,
Niclas Tall,

Examiner: Graham Kemp

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

Modern cars contain fully equipped multimedia (In-Vehicle Infotainment, IVI) systems, which are capable of rendering video to several screens, audio playback to different locations, internet connectivity, third-party applications, indexing media of a user's nomadic devices and more. One of the problems the automotive industry faces today is mining the users' nomadic devices and making the contents available for playback in an acceptable time frame. Current mining software does not fulfill automotive requirements concerning the time between detection of a nomadic device, metadata extraction and subsequent serving of the metadata.

Linux is becoming a popular operating system for IVI platforms. The GENIVI Alliance focuses on developing a standard Linux platform for the IVI industry, and this platform is the target for the Metadex software developed in this thesis.

Since implementing an entirely new mining software would be out of the time frame of this thesis, a survey of existing open source multimedia indexers which could be used as a base for the new miner was conducted. The two candidates with the most appealing features for multimedia indexing in an embedded environment were chosen for evaluation. The candidates were Tracker and Nepomuk-KDE. A benchmark of these two candidates was performed to give a motivation of which candidate the future implementation would be based on.

The resulting software yielded a significant improvement in the time between device detection and metadata availability. The improvements were achieved by indexing metadata in multiple stages where the granularity of the metadata was increased over time.

The results confirm the possibility of achieving fast indexing of nomadic devices which is responsive enough to be used in an automotive environment. We believe that our choice of modifying existing open source software was a fundamental decision that allowed us to implement and meet the set requirements in the time frame of this thesis.

Keywords: automotive, indexing, metadata, tracker, Metadex

Acknowledgments

We would like to thank our thesis advisors Johan Thelin and Graham Kemp for the support and advice given during the thesis. Furthermore we would also like to thank Pelagicore for the opportunity to carry out this thesis.

Jonatan Pålsson & Niclas Tall. Göteborg, Sweden 12/6/2013

Contents

1

Introduction

MODERN cars feature fully equipped multimedia entertainment systems. The *head unit*, and the *Electric Control Units* (ECU) computers of a car are used to not only display GPS maps and car diagnostics, but also for entertainment purposes, such as displaying video, playing music, downloading apps and to browse the internet. A head unit designed for multimedia and entertainment is part of what is called an *In-Vehicle Infotainment* (IVI) platform.

The IVI platform developed at Pelagicore is based on Linux, as standardized by the GENIVI Alliance, and the multimedia indexer developed in this thesis will target this Linux IVI platform.

The IVI platform must be capable of making multimedia accessible and searchable to the user of the system. When the user decides to listen to, or watch a particular song or film, the system must locate this media on any accessible storage and play it for the user. In order to make the multimedia accessible, the media files must be analyzed, and the *metadata* must be extracted and stored in such a way that it can be searched. The process of building a database over metadata gathered from media files will be referred to as *indexing metadata* and the process of file system traversal and metadata extraction is referred to as *mining.* The existing indexing solutions currently available are too slow to be used in an embedded setting. Particularly, the time needed to produce an initial search result, or to determine whether a file is at all available or not, must be kept as low as possible.

The problems this thesis addresses are to lower the time required to produce a first result for the user, and also make the system more responsive where delays are unavoidable.

1.1 Purpose

There are several motivations to quickly index multimedia in cars. Media can not be presented to the passengers of the car if it has not been indexed, and thus a slow indexing process means a worse experience for the passengers than a fast indexing process. The purpose of this thesis is to find heuristics of indexing metadata which satisfy the automotive industry requirements. The currently available multimedia indexing systems were designed with desktop computers in mind, and are not suitable for the less performant embedded systems available in cars.

The main problem with the current solutions is the amount of time required for the indexing process, and this is the focal point of this thesis.

A good user experience is of course important from a marketing standpoint, but it is also important when considering safety. A system which behaves unexpectedly is more likely to capture the attention of the driver, which in turn takes focus from the actual driving, and may cause a safety hazard[1][2].

1.2 Goals

This thesis focuses on improving or replacing an existing metadata mining and service system for multimedia files currently used at Pelagicore AB (see section 2.1 for an introduction of this company).

The current system is based on a single metadata mining sweep, which analyzes each file once, and provides a complete set of metadata gathered from this single sweep. As a potential way of improving the performance of the mining system it is investigated whether it is beneficial for mining speed to split the single sweep into multiple, incremental sweeps. Each incremental sweep will provide additional metadata, and when all incremental sweeps are combined, the output from the sweeps will equal the output from the original single sweep.

Once metadata has been extracted, the purpose of the system is to provide metadata information about the files it has processed. Information is requested from the system by sending queries to it. The hypothesis is that by replacing the single sweep processing with multiple sweeps, the time between when a query was received and when the system can respond will be lowered, since the system will respond to queries as soon as the first stage of mining for a file has completed. Removable flash media is the primary target for indexing. Due to the structure of the most popular file systems for this type of media, it should be possible to perform a first sweep of the most basic data very quickly[3]. As an example, the FAT-family of file systems places information regarding file names, creation and write times in the directory structure[4], which is placed in a known position on the disk and is designed for fast access. By reading only the directory structure during the first sweep, and not the contents of the actual files, performance should be improved over the original single sweep implementation.

The metadata indexing software is used to produce information which is ultimately displayed to users of the IVI system. A hypothesis is that the system appears responsive by continuously updating the user with new progress information as it becomes available[5]. By allowing querists to *register* queries with the multimedia indexing system, querists can be ensured to receive updates as they become available, and display this data as they see fit. This approach of registering queries can be contrasted with the approach where the querist sends multiple queries to the indexing system, and the querist itself decides when to stop requesting new data. By using the proposed subscription model for handling updates, the saturation of the communications channel is likely to be reduced and more timely updates of new query results are expected.

1.3 Method

There are existing systems which perform many of the tasks required by Metadex, and a survey of these systems was performed in order to understand how the existing systems perform these tasks, and which additional tasks may be relevant.

An estimation of the amount of work required for designing a new system was performed, and the amount of work was found to be too high for the time frame of the thesis.

It was investigated if an existing system could be adapted to conform to all the requirements of Metadex. The systems which fulfill the functional requirements of Metadex must also be checked for conformance with non-functional requirements such as licensing and extensibility.

Suitable systems were tested and compared using a series of benchmarks. The most suitable software was selected as a base for the Metadex software.

The technologies used in the selected base software, in addition to the technologies required for adding the selected new features to Metadex, were researched by reading technical and academical reports. A considerable amount of time was spent on understanding the existing code of the base software, and how to properly add new functionality to it.

Modifications and additions to the base software were carried out in a way conforming to the software license of the base software, and in a manner allowing these additions to be sent up stream back to the original project.

1.4 Delimitations

Implementing an entirely new multimedia indexer is an large project in itself. The structure of the stored data is an active field of research[6][7][8], implementing a data store, and schemas based on this research would require an amount of work outside the time frame of this thesis. In addition to the work related to persistence of data, components such as miners, metadata property extractors, directory crawlers must also be developed. Pelagicore is an open source company, and in order to make the software resulting from this thesis as attractive as possible for Pelagicore, the base software candidates considered for Metadex needs appropriate open source licenses. The licenses appropriate for software used in this thesis is discussed in section 2.3.

The choice of base software places restrictions on the programming languages used, and also limit the choices of additional software and libraries to use. The target hardware platform restricts the choice of programming languages and tools even further.

Metadex must run on the Linux platform used at Pelagicore, which means it must make good use of the available libraries and software already present in the IVI platform. The introduction of new dependencies must be kept at a minimum.

1.5 Structure

This thesis is divided into seven chapters. The first chapter gives a broad introduction to the field of IVI and media indexing. Chapter two describes all the technologies used in the thesis, and also gives some extra information regarding different metadata formats, which is useful for implementing metadata extractors (but may be skipped for casual reading).

The third chapter gives a description of the two mining software which were selected as candidates for the basis of Metadex.

In the benchmarks chapter, chapter 4, the two candidates from chapter 3 are benchmarked. In the fifth chapter the implementation of Metadex is described, and the modifications and additions to the software chosen in chapter 4 are described.

The sixth chapter shows the results of the thesis in the form of performance measurements where Metadex is compared to the unmodified base software. Finally chapter 7 gives a conclusion, some remarks and directions for future work.

2 Background

In this chapter the reader will be given the prerequisite knowledge regarding the technologies used and the organizations mentioned necessary to read the rest of this thesis. The topics of the section will start with non-technical aspects of the project and end with more technical topics. Pelagicore, the company supplying the topic for this thesis, will be presented. A brief introduction to open source in the automotive IVI industry is given with the introduction of the GENIVI Alliance. Different licensing issues, and their causes are presented.

A quick introduction to specifying vocabularies is given in the section on ontologies, different types of metadata, and the storage of metadata is presented.

The *GLib* utility library was used extensively in the software described in this thesis. The library is described in this section to make it easier for the reader to understand the structure of the software. Finally the technology used for communicating between different software components is described.

2.1 Pelagicore

The following description is from Pelagicore's website:

> "Pelagicore AB is a Swedish company with offices in Gothenburg, Sweden and Munich, Germany. We are a technology and product development company that focuses on applying Open Source software in the automotive infotainment industry.
>
> Our team of technology and Open Source community experts enable a novel, holistic approach to development with expertise that stretches from silicon design, software as well as User Experience Development to OEM expert advisory roles.
>
> Pelagicore is a key contributor to the GENIVI alliance and several community

> projects such as the Qt Project, Linux kernel and a number GNU/Linux distributions. Based on these community projects we develop automotive Infotainment platforms and key components to accelerate the implementation of customer projects." (About us, pelagicore.com)

2.2 GENIVI

The GENIVI Alliance was established to specify a common software platform for the automotive infotainment industry. The alternative to a common platform is having several proprietary platforms across different vendors. By having a unified platform, the members of the GENIVI Alliance hope to lower development costs and development time for new infotainment systems[9]. The following is a quote from the GENIVI Alliance, detailing their goals:

> "GENIVI® is a non-profit industry alliance committed to driving the broad adoption of an In-Vehicle Infotainment (IVI) open source development platform.
>
> The alliance aims to align requirements, deliver reference implementations, offer certification programs, and foster a vibrant open source IVI community.
>
> Our work will result in shortened development cycles, faster time-to market, and reduced costs for companies developing IVI equipment and software.
>
> GENIVI's objective is to foster a vibrant open source IVI community by:
>
> 1. Delivering a reusable, open source platform consisting of Linux-based core services, middleware, and open application layer interfaces
> 2. Engaging developers to deliver compliant applications
> 3. Sponsoring technical, marketing, and compliance programs."
>
> (GENIVI Alliance, genivi.org)

When using or modifying existing software, it is important to ensure the software licenses of the software are compatible with the use cases of the software. GENIVI has developed guidelines for deciding which licenses are compatible in common automotive scenarios, such as linking proprietary binaries to open source libraries. Several open source licenses have been analyzed and categorized.

The guidelines established by GENIVI are intended to be used when deciding if a particular software is license compatible without having to comprehend the legal aspects of the license in question. The use of an inappropriate license would prevent the software from a potential inclusion in a GENIVI compatible platform.

2.3 Licenses

In this section the open source software licenses encountered during the project are surveyed. The official stance of GENIVI on each of the licenses is given. A common factor among the licenses described here is that they are all *open source* licenses. The basic principle behind open source is the equal rights of all developers contributing to the software project, and the way changes to the software are made available to other developers[10]. The licenses discussed in this section are the GNU GPLv2 and GNU LGPLv2.1, the implications of using these licenses will be discussed in the section below. The GENIVI Alliance has established a compliance program[1] to classify software licenses, the recommendations of this program can be used by members of the GENIVI Alliance. Software licenses are classified into four major groups, which are color coded to indicate the suitability of each license:

- **Yellow-light**: These are the licenses which have not yet been reviewed by GENIVI. Notionally, every license is considered yellow-light until reviewed and categorized by GENIVI.
- **Green-light**: These are licenses which have been reviewed by GENIVI and have been accepted as suitable licenses.
- **Red-light**: These are licenses which have been reviewed by GENIVI and have been rejected as suitable licenses.
- **Orange-light**: These are licenses which have been reviewed by GENIVI and have been accepted as suitable licenses in certain cases.

2.3.1 GNU General Public License Version 2

GNU General Public License Version 2, often abbreviated to GNU GPLv2 or just GPLv2, is one of the most widely used free software licenses[11][12] originally written by Richard Stallman[10].

Software licensed under the GPL is called free software, where free indicates *freedom*, and is not an indicator of price. This section discusses version 2 of the GPL, and it is important to note that there are in fact three versions of the GPL.

Since the release of version 1 in 1989 the GPL has been revised twice, the first revision became version 2 of the license, and was released in 1991. The second revision of the license became version 3, and was written in 2007.

The GPLv2 restricts the ways in which software licensed under it may be copied, distributed and modified. The license is readily available[2], and this text will not repeat the entire license, but highlight the points specifically important for the software developed in this thesis.

- There are no restrictions on executing an application licensed under the GPLv2. This means that a system which is not GPLv2 licensed may execute a GPLv2 licensed application[13].

[1]see: http://www.genivi.org/genivi-compliance-program

[2]For version 2, see: `http://www.gnu.org/licenses/gpl-2.0.html`

- If a new software is derived from a GPLv2 licensed software, this new software must also be licensed under the GPL. The creator of the new software must in some way make the source code, and any build scripts etc. for the derived work available[13].

This license is *Green-lighted* by GENIVI when applied to program code, which means it can be used in GENIVI compatible software projects without further investigation

2.3.2 GNU Lesser General Public License, version 2.1

The GNU Lesser General Public License Version 2.1 (or LGPLv2.1 for short) is a less strict version of GPLv2. The main difference between this license and GPLv2 is that binaries linked to libraries licensed under LGPLv2.1 do not themselves have to be licensed under LGPLv2.1.

If GPLv2 is used instead of LGPLv2.1, all software linked to the library using this license must also be licensed under GPLv2. This act of linking a binary to a library is not seen as *executing the library software*, which is permitted in GPLv2, but it is rather seen as extending the software, which is not allowed unless the derived application has the correct license.

With LGPLv2.1, it *is* allowed to link non-GPL binaries. The use of this license makes sense when aiming to make the library in question useful for organizations or persons who do not use open source[14].

2.4 Ontologies

The data gathered by Metadex is stored in accordance to a set of *ontologies*, defined by the NEPOMUK project. In order to understand what this means, the term ontology must first be defined. Ontologies are used in many different areas, such as for communicating unambiguously between different departments within a company (imagine the Human Resources department and the Salary department using an ontology to establish the meaning of the word *hire*) and for structuring data in a way which facilitates re-use. The term is used both when discussing social interactions as well as what is more interesting in this project, when exchanging and storing data in software[7]. In [8], Thomas R. Gruber defines an ontology as the following:

> "A specification of a representational vocabulary for a shared domain of discourse – definitions of classes, relations, functions, and other objects"
> (Gruber, A Translation Approach to Portable Ontology Specifications)

By using ontologies for specifying the structure of the stored data, data can be shared between different programs, and different components within a program while avoiding ambiguity. The choice of ontologies to present in this section comes from section 3, where the Nepomuk-KDE and Tracker are identified as the two most suitable existing indexing software. Both of these software build on the NEPOMUK ontologies (which have given name to the Nepomuk-KDE project, note however that the NEPOMUK and Nepomuk-KDE projects are different projects). In this section the NEPOMUK ontologies are first presented, since these form a common ground for both projects, and in section 2.4.2 the Tracker ontologies are described as additions and modifications to the NEPOMUK ontologies.

2.4.1 The NEPOMUK ontologies

The NEPOMUK ontologies define the vocabulary and structure for storing data related to the information regarding the users of a desktop computer. Examples of the things described by these ontologies are E-Mails, Instant Messaging conversations, information regarding the physical location of the user of the system, tags embedded in documents and media files, etc.

The ontologies are designed with a general base, and extend to more and more specialized use cases. In [6] the ontologies are depicted as a pyramid. A similar pyramid over the parts relevant to this project is provided in figure 2.1. In this pyramid the top represents the most general components of the ontologies, which is the representation of the ontologies as processed by a machine. The topmost layer is RDF, and a discussion on RDF follows in 2.6. For now, it suffices to say that RDF forms the most basic way of representing data.

In NEPOMUK, the basic features of RDF are augmented by RDF Schemas (RDFS), which allows expressing concepts such as subclassing. If the structure of the ontologies is seen as a pyramid with RDF on top, RDFS follows directly below. NEPOMUK also adds one last layer of augmentation below RDFS to represent named graphs and cardinality, this is called the *NEPOMUK Representational Language* (NRL). In total, RDF, RDFS and NRL form the layer used for representing the ontologies in the machine[6].

The NEPOMUK Annotation Ontology (NAO), NEPOMUK Information Element Ontologies (NIE) and NEPOMUK Graph-Metadata Ontologies (NGM) together form the next level of abstraction. Using only the representational layer there are no capabilities to express high level concepts such as the title of a file, or the language in which a document is written. The NAO, NIE and NGM ontologies add these basic capabilities to NEPOMUK.

As an example of how the ontologies are used, when a music file is to be represented in NEPOMUK, it will contain properties from several layers of the pyramid. The NEPOMUK MultiMedia Ontology (NMM), which is yet another level of abstraction away from the representational layer, provides an artist property, which is set to the artist of the music piece. NIE contains a mime-type property which is set to the actual file type of the music file and NAO has a property for the modification date of the music file.

2.4.2 The Tracker ontologies

The Tracker ontologies are based on the NEPOMUK ontologies, and contain some extensions and fixes. Some of the changes introduced relate to new platforms, such as properties regarding physical locations required by the MeeGo platform. Other changes

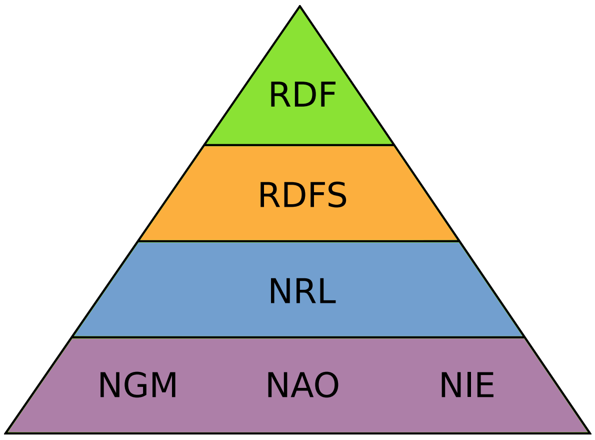


Figure 2.1: A pyramid depicting the relevant relationships inside the ontology structure of NEPOMUK, heavily inspired by figure 13.2 of [6]. The generality of the layers decreases from top to bottom. The complete layout of NEPOMUK contains more layers, which are not relevant here, and have been left out for brevity.

relate to the incorporation of other metadata standards, an example of this is the addition of the *Dublin Core* (see section 2.5.1) Ontology.

The addition of this ontology means data can be categorized in a uniform way across different resource libraries.

Tracker also adds properties for handling removable media, whether devices are currently connected, whether data can be written back to the resource in question (in contrast to only reading from the resource). These properties are placed in the TRACKER ontology.

2.5 Metadata

The term *metadata* comes from the Greek term *meta* which means *among*[15], metadata refers to *data about data*. The National Information Standards Organization (NISO) describes three subcategories of metadata, with different purposes. *Structural metadata*, which, when describing digital objects details the design and the specification of data structures and information data containers, as an example, structural metadata regarding physical books could detail which pages are contained in a specific chapter.

Descriptive metadata facilitates discovery of the objects it describes, this data is commonly used to locate objects in catalogs or via searching. Descriptive metadata describing a musical recording could detail properties such as artist, title or genre of the recording. This sort of metadata, together with the last sort, which is known as *administrative metadata* is what Metadex focuses on. Administrative metadata describes properties such as where a file is located in a file system, or which access rights are required to access the file. Administrative metadata for a physical book in a library could detail the shelf which holds the book, for example[16].

This section will present important standards for metadata, and discuss how they differ.

2.5.1 The Dublin Core

Dublin Core is the name of a set of 15 core properties used to describe resources. The properties were defined by the Dublin Core Metadata Initiative (DMCI), and are typically used by projects handling large amounts of data, such as the MusicBrainz online music encyclopedia project, or by libraries[3][17].

The specification of Dublin Core started in 1995 with an invitational workshop in Dublin, Ohio[17]. In contrast to the following metadata formats in this section, Dublin Core does not specify how this data should be represented, but only offers the set of possible properties and their semantics.

The properties are frequently used as a base for other formats, which is extended with more specialized properties as needed by the implementing software. Listing 2.4 shows how Dublin Core (the `dc` prefix) is used to represent information regarding a music file, and in this example it is augmented by the NEPOMUK metadata properties.

2.5.2 Advanced Systems Format

The ASF container format was designed and is maintained by Microsoft Corporation. The purpose of the format, which was originally called the *Advanced Streaming Format*[18], is to provide efficient playback for streamed multimedia as well as local playback. The container format does not specify which codecs (software for coding and decoding data streams) it should contain, but is known to wrap the *Windows Media Video* (WMV) format as well as *Windows Media Audio* (WMA).

ASF is composed of a series of objects. Some of these objects contain information which must be read before playback of the actual media can commence, such as stream properties or file properties. In order for the format to support streaming media, it is important that properties such as encryption information and bit rate are present before the media is played, and therefore this type of information is placed first in the file or stream.

The ASF format contains several objects with metadata information, distinctions are made for metadata relating to a specific stream (as the ASF file can contain several streams), information regarding branding of the data (such as banner images), stream specific data, and metadata specific to a certain language[19]. The container can store arbitrary key-value metadata pairs, and the specification gives advice on which object to place specific key-value pairs in.

[3]A comprehensive list of projects using the Dublin Core can be found at: `http://dublincore.org/projects/`

2.5.3 Ogg

The purpose of the Ogg container format is to create an open standard for carrying encoded media such as audio and video. In contrast to other popular formats, there are no licensing fees involved when using Ogg.

The Ogg *Request For Comments* (RFC) document, which forms the official specification, does not specify support for metadata other than data detailing technical properties of the streams the Ogg contains. There is no support for content metadata such as the author of the contained streams. In order to specify the metadata to go with an Ogg container, the metadata must be added via one of the data streams which the Ogg contains[20].

The most popular formats to put in an Ogg container, which are also mentioned in the RFC, are Vorbis and Theora. In order to describe how metadata is handled in Ogg it is therefore more appropriate to describe metadata using these two formats.

The Vorbis and Theora specifications both describe comment headers, which in practice are used for storing metadata, the specifications advise against this however:

> "The Vorbis text comment header is the second (of three) header packets that begin a Vorbis bit stream. It is meant for short text comments, not arbitrary metadata; arbitrary metadata belongs in a separate logical bit stream (usually an XML stream type) that provides greater structure and machine parsability.
>
> The comment field is meant to be used much like someone jotting a quick note on the bottom of a CDR. It should be a little information to remember the disc by and explain it to others; a short, to-the-point text note that need not only be a couple words, but isn't going to be more than a short paragraph" (The Vorbis I specification[21])

The Theora specification contains a similar section[22]. There is an emerging format, *Ogg Skeleton*, designed to provide a metadata bit stream to be used instead of the comments section in Theora and Vorbis. As of this writing it is, as mentioned, common to place metadata in the comments headers of the Vorbis and Theora streams, and it is this data which is parsed by metadata parsers supporting Ogg with Vorbis or Theora.

2.5.4 MPEG-4

MPEG-4 is an open international standard which provides tools for delivering multimedia, these tools form a multimedia framework for audio, video, graphics with or without interactive features. The standard also includes codecs for video and audio encoding and decoding, namely the Advanced Video Coding (AVC) and the Advanced Audio Coding (AAC) codecs. The MPEG-4 standard is divided into 30 parts, where part 3 describes audio[23] codecs, part 12 describes the ISO Base Media File Format details about the storage of timed media information[24], such as motion pictures and audio streams, and part 14 describes the MP4 file format[25] - which is the container format described in this section.

An MP4 container may include metadata by embedding the file structure defined by MPEG-4 Part 12. The MPEG-4 Part 12 file structure was designed in an object-oriented manner, where a file can be decomposed into constituent objects, and the structure of the object can be inferred by its type, which yields fast processing. An MPEG-4 Part 12 file is composed of a series of objects, called boxes, which may be nested. The sequence of boxes in the file shall contain at least one outermost wrapping box, and it is usually located close to the beginning of the file or in the end of the file. The wrapping box contains further, possibly nested, boxes describing properties such as MIME-type of the metadata (such as XML metadata, binary metadata), and the actual meta data. An example of such boxes is the 'xm' and 'bxml' boxes, which can contain metadata described by XML.

2.5.5 Exif

The Exif format specification was first released in 1996, by Japan Electronic Industry Development Association (JEIDA), and has since become a very popular format for storing metadata in Digital Still Camera (DSC) files. The standard was developed with the intent to ensure data compatibility and exchangability in DSC (and to a smaller extent also audio) files[26].

The tags in the Exif specification are based on the tags from the *Tagged Image File Format* (TIFF), The Exif tags are a superset of the TIFF tags. The original TIFF tags include properties such as the size in pixels of the image, the software used to create it and information regarding the copyright holder. Exif extends the TIFF tags with information regarding the conditions in which the photography was taken, such as the aperture, exposure time and ISO speed settings of the camera.

Listing 2.1 contains an example of the metadata stored in a JPEG file using Exif, as presented by the MediaInfo tool[4].

Exif data is designed to be embedded in the DSC file it describes. The Exif specification describes how Exif data is embedded in JPEG and TIFF files. In order to maintain compatibility with the original JPEG specfication, Exif places its data in the second JPEG *application segment*, APP1 (there is also a 0:th segment). The JPEG specification allows several 64kB application segments[27], and Exif can extend to multiple of these segments in order to store all available tags. The APP$_n$ segments are located near the beginning of the JPEG file.

TIFF uses a list of pointers (called tags) to identify data sections of the file. These data sections may contains actual image data, or optionally application specific data. To store Exif information in TIFF files, specific TIFF tags reserved for identifying Exif data are used to indicate the offset to data sections of the file which contains the Exif data. The list of tags is located near the beginning of the file, but the actual Exif information is not necessarily located in the beginning of the file [28].

The actual Exif data stored in both JPEG and TIFF files has the same form. Each Exif tag has a numeric identifier and a specified parameter length and type. ASCII pa-

[4]MediaInfo is available from: `http://mediainfo.sourceforge.net`

Listing 2.1: Example Exif data

```
File name        : Helsingborg_fortress2.jpg
File size        : 650392 Bytes
MIME type        : image/jpeg
Image size       : 2048 x 1536
Camera make      : OLYMPUS OPTICAL CO.,LTD
Camera model     : X200,D560Z,C350Z
Image timestamp : 2005:07:20 15:38:21
Image number     :
Exposure time    : 1/800 s
Aperture         : F3.8
Exposure bias    : 0 EV
Flash            : No, auto
Flash bias       :
Focal length     : 10.0 mm
Subject distance:
ISO speed        : 64
Exposure mode    : Auto
Metering mode    : Multi-segment
Macro mode       : Off
Image quality    : Standard Quality (SQ)
Exif Resolution : 2048 x 1536
White balance    : Auto
Thumbnail        : image/jpeg, 5891 Bytes
Copyright        :
Exif comment     :
```

rameters are variable in length, but must instead be NULL-terminated. As an example, Exif data detailing image width is identified with `0x0100`, and immediately following is a numeric value containing the width of the image in pixels[26].

2.5.6 IIM and XMP

In early 1990 International Press Telecommunications Council (IPTC) developed the *Information Interchange Model* (IIM) data model for universal communication for different types of data including, text, photos, graphics and audio. The software company Adobe later adopted a subset of the IIM properties for the Photoshop imaging software, which became the first widely used program to place metadata in images[29].

In 2001, Adobe introduced a new metadata technology known as *Extensible Metadata Platform* (XMP) which was inspired by the subset of IIM properties used in Photoshop. This new format is used to embed data into the file itself or to be placed in a separate file. XMP standardized the definition, creation, and processing of metadata, by providing a data model, a storage model and schemas. In the storage model, metadata consists of a set of properties, where a property is always associated to an entity referred as a *resource* and the property describes this resource. A resource may be a JPEG file or a document file such as a PDF document. A property consists of an identifier and a value, which makes a statement about the resource[30]. In order to represent the metadata properties, XMP makes use of the Resource Description Framework (RDF) standard, which is described in section 2.6. RDF is an XML based technology developed by World-Wide Web Consortium (W3C). Listing 2.2 contains an example of an resource (a PDF document) with two properties. To embed XMP into a file it must be wrapped in an *XMP Packet* containing a header, serialized XMP, padding and ending trailer. The packet is embedded differently depending on the file format. XMP gives official guidance for embedding into the following formats: TIFF, JPEG, JPEG2000, GIF, PNG, HTML, PDF, AI, SVG/XML, PSD, PostScript, EPS, and DNG. For JPEG the XMP Packet shall be located in the designated *APP1* segment, which is the same as for Exif, as was discussed in section 2.5.5.

XMP also supports external storage of the metadata, in a so called *sidecar file*, which may be necessary for unsupported file formats, access issues and etc. The sidecar files must consist of well-formed XML.

Several standard metadata representations can be expressed in XMP, such as Dublin Core, Exif, IPTC Core, and IPTC Extension.

Listing 2.2: Serialized XMP properties in the RDF format

```
<x:xmpmeta xmlns:x='adobe:ns:meta/'>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 <rdf:Description xmlns:dc="http://purl.org/dc/elements/1.1/">
   <dc:format>application/pdf</dc:format>
 </rdf:Description>
 <rdf:Description xmlns:xmp="http://ns.adobe.com/xap/1.0/">
   <xmp:CreateDate>2002-08-15T17:10:04Z</xmp:CreateDate>
 </rdf:Description>
</rdf:RDF>
</x:xmpmeta>
```

2.5.7 ID3v2

A very common metadata format for MP3 files is the ID3v2 format. As the name suggests, an ID3v1 format also exists, but has been superseded by version 2, and only ID3v2 will be covered in this section.

The ID3v2 3.0 Informal standard[31] serves as the specification for ID3v2. ID3v2 metadata tags start with a four character ASCII identifier, followed by the size of the actual data, and flags indicating whether the data is compressed, read only, encrypted or has some other property[31]. A file may contain up to 256MB of ID3v2 metadata (limited by the 28bit size field in the ID2v3 header), and the metadata is placed first in the file it describes.

Since MP3 does not have built in support for non-audio data, it is important to ensure the MP3 decoder does not attempt to play the metadata as if it was audio, since this would most likely only produce noise. The decoder looks for a synchronization signal, a special sequence of bits, in the data stream, and if one is found the data is determined to be audio. The ID3v2 metadata should normally not contain this signal, but if it does, this signal must be removed when writing the metadata to the MP3 file[31].

Listing 2.3 shows example of the ID3v2 metadata extracted from an MP3 file using the `id3v2` tool[5].

Listing 2.3: Example of ID3 metadata

```
ID3v2
TALB (Album/Movie/Show title)              : American Life
TIT2 (Title/songname/content description) : I'm So Stupid
TPE1 (Lead performer(s)/Soloist(s))        : Madonna
TPUB (Publisher)                           : Maverick
TLEN (Length)                              : 9240
TYER (Year)                                : 2003
TDAT (Date)                                : 0421
TSSE (Software/Hardware and settings)      : LAME v3.97
COMM (Comments)                            :
```

2.6 Storage and querying

The *Resource Description Framework* (RDF) is a language used to describe entities and relationships as graphs, using URIs as identifiers and by setting properties for each entity [32]. RDF is a W3C recommendation, and is maintained and specified by the W3C. RDF can be represented using a language such as XML. While it is convenient for humans to consume an RDF description in the form of a graph, it is more convenient for a machine to use *triples* for representing data. Any RDF description can be expressed equivalently as a graph and as a set of triples[32], which is very useful when having a machine store the data.

The RDF language is very flexible and extensible, and is therefore a good way to represent metadata, especially when different sorts of media require different properties. By the design of the ontologies (see section 2.4) the different media types are clearly separated, and this separation is used in the RDF representation as well.

In an abstract form, RDF triples often consist of the following fields: `<subject>` `<predicate>` `<object>`.

In the example listing 2.4, the music file is described using elements from the Dublin Core specification (see 2.5.1), as well as elements from the NEPOMUK ontologies (see 2.4.1). While this example is not optimal in regards to the data specified, it serves well to show how different ontologies can be mixed to specify the properties of an element.

While RDF is a suitable format for storing the data, it is not possible to construct a query for the data of a specific element or field using RDF. In order to create queries for data stored in RDF the *SPARQL Protocol and RDF Query Language* (SPARQL)

[5]See `http://id3v2.sourceforge.net/` for more information on the `id3v2` tool

Listing 2.4: An example of an RDF description of a music recording

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:nfo="http://www.semanticdesktop.org/
            ontologies/2007/03/22/nfo/">
  <rdf:Description>
    <dc:creator>House of Pain</dc:creator>
    <dc:title>Jump Around</dc:title>
    <dc:date>1992-11-01</dc:date>
    <dc:language>EN</dc:language>
    <dc:publisher>XL Recordings</dc:publisher>
    <nfo:channels>2</nfo:channels>
    <nfo:fileUrl>file:///music/a64818c1b7830.mp3</nfo:fileUrl>
  </rdf:Description>
</rdf:RDF>
```

language is used. SPARQL is, just as RDF, a World-Wide Web Consortium recommendation, and the W3C maintains and has specified SPARQL[33].

In listing 2.5 is an example of a SPARQL query, which will return the title and URL of the file described in the RDF example above. The `WHERE` clause is matched with the elements in the RDF database, and the fields specified in the `SELECT` clause are returned to the querist.

Listing 2.5: SPARQL query to extract title and URL for a subset of files

```
SELECT  ?title nfo:fileUrl(?x)
WHERE   { ?x dc:title ?title
          FILTER REGEX(?title, "^Jump") .
          ?x dc:language "EN"
        }
```

There are different ways to store the actual data represented by the RDF expressions. A common method of storage, which will also be used in this project, is to convert the RDF specification in to a suitable schema for a relational database. When executing SPARQL queries on the data, the SPARQL queries are converted to SQL queries for the SQL database, and the query is executed as an SQL query.

2.7 GLib

GLib was originally included in the *GTK+ toolkit*[6], which is used to create graphical applications for the *GNOME*[7] desktop system. GLib provided the data structures and

[6]See `http://www.gtk.org/`

[7]See `http://www.gnome.org`

other features not relating to graphics in GTK+, and was eventually separated from the rest of the graphical toolkit in order to be more attractive for non-graphical applications.

GLib in combination with the C language can be compared with the Standard Template Language (STL) in combination with C++, or the Qt framework[8], with which it shares many similarities. GLib provides the C programmer with support for advanced memory allocation, data structures such as lists, trees and tables, and using GObject there is also support for object orientation[34].

This section can be used as a reference, and will be referred to when discussing implementation details which make use of GLib specific features.

2.7.1 GObject

GObject provides mechanisms for memory allocation of classes, inheritance and linking parent-child relationships between classes, as well as signaling. Working with GObject is very similar to working in an object oriented language such as C++ or Java, with the main difference being that the object orientation is not included in the syntax of the language (in our case this is C).

Some examples of where GObject syntax is unfamiliar for C++ or Java programmers include constructors, member functions, class instantiation, and signaling between objects.

Class instantiation

In C++ a class is instantiated, and space is allocated for it on the heap, using the `new` keyword:

```
MyObject *myObject = new MyObject("Value␣for␣parameter␣1");
```

Since there is no `new` keyword in C, class instantiation in GObject is performed by calling a regular function instead. There are different ways to instantiate classes in GLib. The standard way of instantiating does not allow the initialization of the object to fail. In C++ you might throw an exception in the constructor of the class to indicate that initialization failed, but there are no exceptions in GLib, so failure must be indicated differently.

As can be seen in listing 2.6, the second method for instantiating the class allows the programmer to handle errors which may occur. Also of interest is the difference in how parameters are handled. In C++ parameters are passed to the constructor in a specific order, and are fetched in the same order. GLib uses the notion of *properties* instead. Properties are passed as key-value pairs (every two parameters to the instantiation function after the type are a pair). The properties are then installed in the constructor, and the object is instantiated with the proper values.

[8]See `http://qt.digia.com` for more information on the Qt framework

Listing 2.6: Different ways to instantiate a GObject

```
// Standard instantiation:
MyObject *myObject = g_object_new(MY_OBJECT_TYPE,
                              "parameter1", "Value␣for␣parameter␣1",
                              ...);
// myObject is guaranteed not to be NULL

// Accounting for failure in initialization:
MyObject *myInitableObject = g_initable_new(MY_OBJECT_TYPE
                              "parameter1", "Value␣for␣parameter␣1",
                              ...);
if (!myInitableObject){
  // Handle error
}
```

Constructors

In C++ the constructor bears the same name as the class it constructs. The constructor also has a fixed set of parameters which the programmer can specify:

```
class MyClass {
 MyClass(int x) { this->x = x; }
}
```

GObject has several layers of constructors, the newly constructed object must itself create a chain to the appropriate parent. Due to the way parameters are passed, it is not possible to use the actual parameter values inside the constructors. The purpose of the constructors is to initialize the object, and the parameter values are set after the code in the constructors has executed.

Member functions

In C++ the programmer may use either the pointer operator (->) for accessing fields of a heap allocated object, or the dot operator (.) for accessing fields of stack-allocated objects:

```
MyObject *myObject = ...;
myObject->myFunction();
```

The same syntax would in theory be possible in GLib if classes were treated as structs, but the more common syntax is to prefix the function name with the name of its class and pass the instance of the class to it:

```
MyObject *myObject = ...;
MyObject_MyFunction(myObject);
```

2.7.2 Signals

Signals in GLib are a general purpose notification mechanism. A common use case of signals is to have an object expose one, or several signals, identified by textual names. By exposing these signals, the object indicates that these signals can be used by the object to indicate the occurrence of certain events. Consider the following example, loosely based on crawling in the Tracker software.

> A file indexer, `FileIndexer`, object exposes a signal named `FileFound`, the signal carries a string parameter by the name `FileName`. This signal is *emitted* by the file indexer object each time it finds a new file in the file system. An object interested in being notified when new files are found in the system, such as a `MetadataExtractor` may *register* with the file indexer in order to receive a notification when this signal is emitted. Signal emission is notified in the `MetadataExtractor` by having `FileIndexer` run a callback method on the extractor object during emission.

The callbacks registered with a specific signal are executed in a known order, and new callbacks can be added anywhere in this order. Signals can also be blocked, and unblocked in order to temporarily disable notification of the objects registered.

Signals are inherited in the GObject class hierarchy, which means that signals introduced for a parent type are also available for derived types. Signals are created using `g_signal_new(..)` located in `Signals` module. Objects register callbacks using `g_signal_connect(..)`, providing a reference to the object emitting the signal, and a string identifying the signal. To emit new events, the emitting object calls `g_signal_emit_by_name(..)`, providing the signal identifier. The code in listing 2.7 shows how an object `obj` may create and emit a signal. There is no feature in C++ equivalent to the signal handling of GLib, and thus no example is available for C++.

Listing 2.7: An object `obj` creating and emitting the signal `signal_identifier`

```
/* The following is typically placed in the initialization
 * process of the object */
g_signal_new ("signal_identifier",
              G_TYPE_FROM_CLASS (object_class),
              G_SIGNAL_RUN_LAST,
              G_STRUCT_OFFSET (MetadexMinerClass, metadex_finished),
              NULL, NULL,
              metadex_VOID__VOID,
              G_TYPE_NONE,
              0);

/* The following is executed once an expected event has
 * occurred, and the signal should be emitted */
g_signal_emit_by_name (obj, "signal_identifier");
```

The code in listing 2.8 shows how an object register for events on the signal `signal_identifier` emitted by `obj`. The `G_CALLBACK` macro is simply a void pointer type cast.

Listing 2.8: An object registers for events on the signal `signal_identifier`

```
/* signal_identifier_cb is a callback function defined in
 * the same class as this code is executed */
g_signal_connect (obj, "signal_identifier",
                  G_CALLBACK (signal_identifier_cb),
                  NULL);
```

2.8 D-Bus

D-Bus is a system for *inter-process communications* (IPC), and is used to communicate between different applications over a common communication bus. D-Bus can be likened to communication over UNIX sockets (which is commonly used as a transport for D-Bus), but has some additional features which will be outlined in this section. The messages are represented in a binary format, and overall communication over D-Bus is fast. The D-Bus specification is maintained by freedesktop[9], which also provides reference implementations, tutorials and documentation.

In contrast to UNIX sockets, communication over D-Bus is brokered by a daemon. D-Bus supplies both a system wide daemon, communicating on the *system bus* and a per-user-login-session daemon, communicating on the *session bus*. The system daemon is used to communicate events regarding the actual system, such as when new hardware has been added (removable media, USB sound cards, etc), changes to the print queue, or other events relevant to all users of the system. The session bus is used for general IPC needs for the applications of a single user, examples here include received instant messages, or interactions the user performs in the user interface. Interested applications can register to receive messages on both the system and session bus, and respond to the messages in an appropriate way, for example when all media on a removable device have been fully indexed, an on-screen display software may pop-up a message box.

For the purposes of this thesis there are a few different types of messages commonly exchanged over D-Bus, basic messages with a single recipient, and broadcasted messages which can be intercepted by any interested party. A basic message can be used to create *remote procedure call* (RPC) functionality, both synchronous and asynchronous.

All messages sent over D-Bus must be correctly typed according to the D-Bus type system, and communication endpoints have identifiers which are guaranteed to be unique, both of these features are improvements when comparing D-Bus to UNIX sockets.

The D-Bus low-level API reference implementation has been heavily tested in the real world over several years, and is now considered stable[35].

[9]`http://freedesktop.org`

3

Previous work

There exists a wide range of different media indexing software. The software of interest for this thesis are capable of running on the Linux platform and their licenses are compatible with industry policies and requirements. Below is a list of the most interesting software, and a short description of each indexer. Finally in this chapter, two indexers, Tracker and Nepomuk-KDE, are analyzed in detail, since these two indexers are candidates to be used in the project.

- **Strigi** is a file indexer with *Full Text Search* (FTS) capabilities, provided by the Lucene/CLucene search engine. Clients communicate with Strigi using D-Bus, and the software is written using the Qt framework. Both of these properties make this project interesting for us. Strigi builds an index over all files in the directories specified, and allows searching within their contents. This indexer does not appear to make all the connections between files which we need, such as which audio files belong to a specific album. The FTS capabilities are fast, but this is not the most important feature for Metadex.

- **Beagle** is a search tool designed primarily to index text files, such as emails, Instant Messaging conversations and web pages. The project was developed in C# using a C# port of the Lucene FTS engine[36]. The project is now unmaintained and development appears to have ceased around 2008. Since Beagle was primarily designed for indexing text, is now unmaintained and uses C# (which would require us to ship the Mono platform[1]), we decided against using Beagle.

- **Docfetcher** is, like Beagle, aimed at indexing text documents. The software is developed using Java, but it unlike Beagle still maintained. Java is not currently available in the target system, and adding it with our system would mean a considerable overhead. Because of these undesirable properties of Docfetcher, we decided against it.

[1]See `http://www.mono-project.com`

- **Recoll** is, like Beagle and Docfetcher, aimed at documents. This is the primary reason for us not to use it.

- **Terrier** is a large scale *information retrieval* (IR) system, developed by University of Glasgow with the purpose of being used as a test bed for new IR applications. The system in itself is designed to be run in a distributed environment, and much focus is placed on MapReduce[2] systems, which is not applicable in our embedded system.

- **Nepomuk-KDE** is the desktop search engine for the KDE desktop environment. This project is capable of indexing multimedia data as well as text documents. The multimedia indexing capabilities of Nepomuk-KDE are suitable for our project. Other factors, such as the tight coupling with KDE libraries, and the overall integration in the KDE system means KDE would either need to be separated from Nepomuk-KDE, or shipped with our final product.
  Separating KDE from Nepomuk-KDE seems to be a very time consuming task, due to the tight coupling with the KDE libraries and will not be attempted. Nepomuk-KDE is one of two main candidates in our choice of media indexing software, however, as benchmarks show later in this report, in chapter 4, Nepomuk-KDE is outperformed by the Tracker system, which does not have the same coupling problems to a large software platform.

- Finally, there is the **Tracker** software which is usually shipped as the desktop search engine for the GNOME desktop. Tracker was designed with embedded systems in mind, it has successfully been used in mobile platforms, such as MeeGo[3]. The metadata miner is capable of handling multimedia data and text documents, just like Nepomuk-KDE, Tracker uses a version of the NEPOMUK ontologies to describe the relationships between different entities.
  The choice of libraries and programming language makes the software portable and suitable for our systems. A disadvantage of Tracker in its current state is that it fully extracts metadata from each file before making it available for searching. This makes the time to first search result high, and gives the impression of the search being slow.

  Tracker is the system currently in use at Pelagicore, and has previously been the software recommended for multimedia indexing by the GENIVI organization.

3.1 Tracker

Tracker is the current system, which Metadex should replace. The Tracker D-Bus API is currently used to communicate with Tracker and there is already code supporting this in the platform developed by Pelagicore. The system has many of the features

[2]See `http://research.google.com/archive/mapreduce.html`
[3]See `https://meego.com/`

desired in the final product, and was therefore chosen as the base of this project. In this project the actual implementation consists of a modified version of Tracker where current shortcomings are mitigated, and missing features have been added.

This section outlines the structure of the original Tracker software which this thesis is based on. The Tracker software is described in increasingly fine grained detail, starting with the runnable applications, to finally describe the different subsystems and modules of the software on a source code level. Parts of the software which are not relevant to Metadex have been left out for brevity where possible.

3.1.1 Overview of applications

Tracker is modular in its design. Seen from the operating system, Tracker is composed of several different processes communicating over D-Bus (see section 2.8). The most important processes, for this project are the following:

- `tracker-control`, which is used to send controls signals to the other Tracker processes. This program can be used to send the `SIGKILL` or `SIGTERM` signals to the Tracker processes, thereby terminating the processes. Stop and start signals can be sent to miners over D-Bus, and the database can be managed (cleared for instance). `tracker-control` is a runnable binary, and is typically invoked via the shell, or by a process.

- `tracker-miner-fs` is the original miner process, this is completely replaced by Metadex, which will be described later in this report. The `tracker-miner-fs` process is responsible for coordinating crawling, extraction and database communication when mining metadata from files in the system. The process is typically run as a long-lived daemon, which indexes new media as it becomes available to the miner. The process is typically not interfaced with, except for starting, stopping and pausing and can be run both by triggering it with a D-Bus request or via the shell.

- `tracker-extract` is used as a common interface to the different metadata extractors which provide the miner processes with metadata. This process is typically invoked via D-Bus, but can also be run from the shell for diagnostic purposes. A typical use case is for the miner to send a D-Bus message to the `tracker-extract` process with a URI for a media file, and receive a SPARQL fragment which can be used when building a SPARQL query for inserting the metadata into the database.

- `tracker-sparql` is used to query, or change data in the Tracker database. This command is mostly used for diagnostic purposes, as the store is normally queried directly over D-Bus. This command is invoked on the shell.

- `tracker-store` is the front-end to the actual database software used in Tracker. Currently this is the SQLite[4] database. Tracker store accepts D-Bus commands

[4]See `www.sqlite.org` for a description of the SQLite project

with SPARQL queries for querying and modifying the database. Since the queries are expressed in SPARQL and not SQL which is the native language of SQLite, the queries are first translated from SPARQL to SQL before being executed against the database.

Each of the Tracker processes builds upon several core libraries. In order to more easily discuss the modifications to the Tracker system, the following section will give a high-level explanation of how the important library `libtracker-miner`, which is responsible for indexing and mining, is designed architecturally. Much of the communication inside Tracker is carried out using signals, see section 3.1 for a comprehensive graph over the signals used in Tracker.

- `IndexingTree` is responsible for keeping track of the directories to be indexed. Internally it builds a *rose tree* data structure, where each node can have arbitrary many children, this mimics how a file system is structured, where a directory can have arbitrarily many sub directories. `IndexingTree` defines three signals; `directory-added`, `directory-removed`, and `directory-updated` which signal changes of the indexing tree to the `FileNotifier`, which is registered as a listener for these signals.

- `Crawler` is used to crawl directories instructed by `FileNotifier` via the `tracker-_crawler_start(..)` function. Internally it keeps a double-ended queue on which directories to crawl.

- `Monitor` is responsible for setting up new monitors for directories given through `tracker_monitor_add(..)`, typically called by the `FileNotifier`. The actual set up of the monitor is done by the GLib module `GFileMonitor`, which is an abstraction layer above *inotify*[5], `kqueue`[6], `FAM`[7], etc. The `Monitor` is registered to the `changed` signal emitted by the `GFileMonitor` object, this signal is then handled, and the appropriate signal is emitted for further handling by the `FileNotifer`.

- `FileNotifier` is responsible for handling events that occur in the indexing tree, updates from the crawler, and creation/deletion/modifications to files in the file system monitored by the `Monitor`. The `FileNotifier` is the module tying `IndexingTree`, `Crawler` and `Monitor` together by registering to their signals. For example when `Monitor` detects that a directory has been created, it emits `item-created`, `FileNotifier` receives the signal emitted by `Monitor` and instructs the `Crawler` to crawl the directory supplied by the signal for more files.

- `TrackerMiner` is an abstract base class to help developing data miners for `tracker-store`. Since it is an abstract class it only provides common functionality for

[5]For a good introduction to *inotify*, see the inotify man-page of the Linux Programmer's Manual (`man 7 inotify` on most recent Linux systems)

[6]For a good introduction to `kqueue`, see the kqueue man-page of the OpenBSD Programmer's Manual (`man 2 kqueue` on most recent OpenBSD systems)

[7]See the `FAM` project page for more information: `http://oss.sgi.com/projects/fam/`

implementing miners, such as setting up D-Bus and GLib signals for controlling miners.

- `TrackerMinerFS` is an abstract class for file system miners which collects data from the file system. Since it inherits from `TrackerMiner`, it has access to its parent properties. `TrackerMinerFS` abstracts away the crawling, monitoring and the communication to the back-end database manager, leaving objects that inherit this class to decide which directories and files should be processed and it is this class the Metadex miner will inherit from.

The graph of figure 3.1 displays a high-level flow chart of the signals between the different modules:

3.1.2 The TrackerMiner and TrackerMinerFS abstract classes

In order to gather files from the file system, the Tracker project provides an abstract base class called `TrackerMinerFS`, which contains functionality such as crawling and creating data structures containing the files which have been or will be mined.

The miners written for Tracker are based `TrackerMiner` abstract class. This class provides the most basic features which any miner should have. Much of the internal communication within Tracker happens over signals, and it is crucial that these signals are well known and standardized, otherwise emitters and listeners will be using different signals, and communication is impossible.

The `TrackerMiner` abstract class provides these standard signals, and also sets up some basic D-Bus listening functionality, such as probing for status, starting and pausing miners.

The `TrackerMinerFS` abstract base class is more complex and has more features than the `TrackerMiner` class. This class builds on top of `TrackerMiner`, and provides functionality for extracting data from file systems.

Via this class, implementors gain access to a file system crawler, an abstraction of the file system, built on top of `GFile`s from GLib, an indexing tree of files to be indexed, and more.

The typical use case of this class is to add the directories to be indexed to the indexing tree somewhere early on in the implementing class, and then later on signal for the miner to be started, which will trigger the file system crawling process.

The file notifier receives callback events from the crawler when a file or directory is crawled. The file notifier checks whether the file or directory is indexable, and notifies the crawler. Once the crawling has finished completely, the file notifier processes the data structures gathered by the crawler. The notifier decides whether to pass control along to the `TrackerMinerFS` class based on whether the file in question is already indexed, and whether it has been modified since the last indexing.

The `TrackerMinerFS` class has registered callback functions with the file notifier, and when a file is either created, moved, deleted or updated, one of these callbacks is called, and control is transferred back to `TrackerMinerFS`.

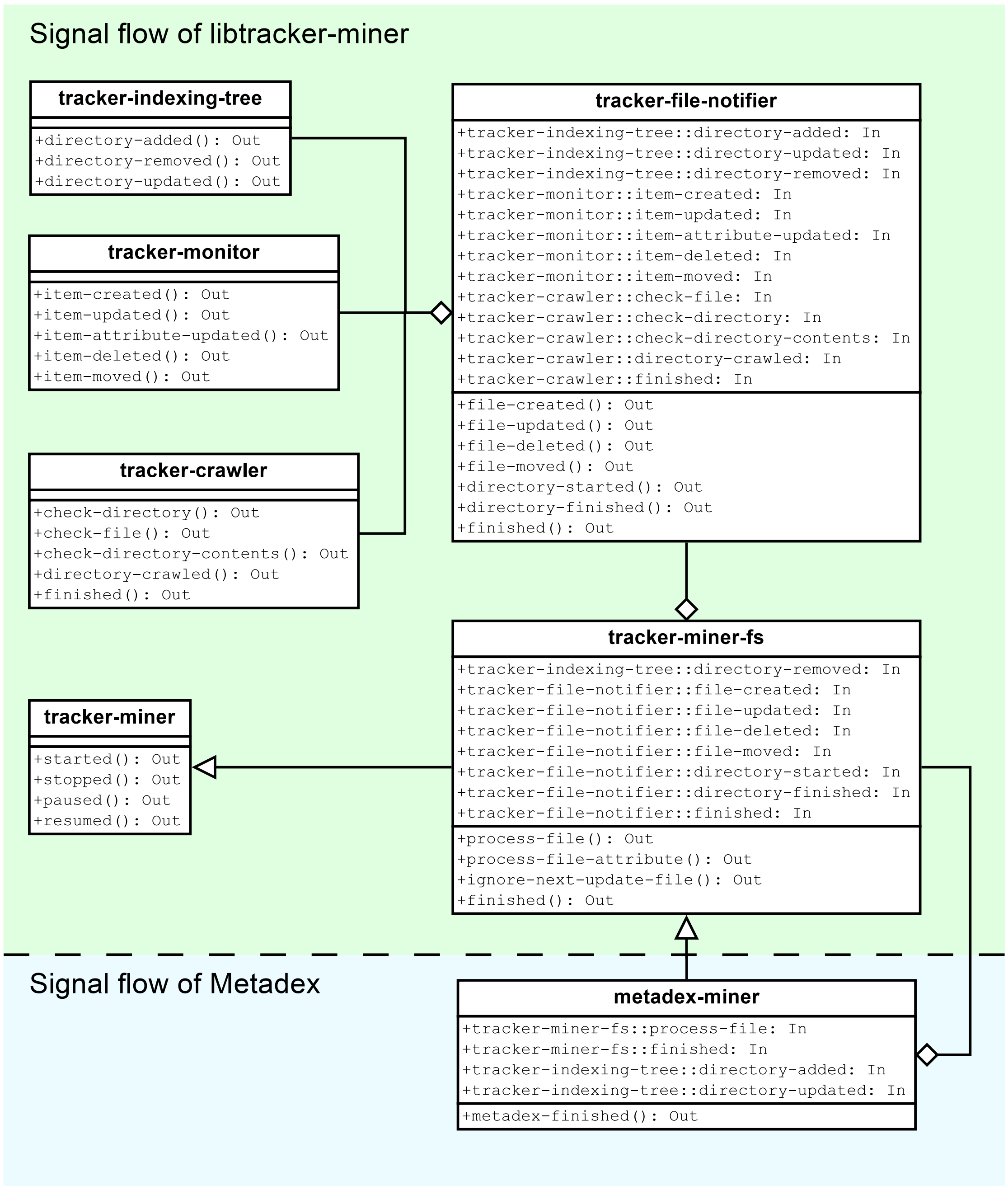

Signal flow of libtracker-miner
tracker-indexing-tree
+directory-added(): Out
+directory-removed(): Out
+directory-updated(): Out
tracker-monitor
+item-created(): Out
+item-updated(): Out
+item-attribute-updated(): Out
+item-deleted(): Out
+item-moved(): Out
tracker-crawler
+check-directory(): Out
+check-file(): Out
+check-directory-contents(): Out
+directory-crawled(): Out
+finished(): Out
tracker-file-notifier
+tracker-indexing-tree::directory-added: In
+tracker-indexing-tree::directory-updated: In
+tracker-indexing-tree::directory-removed: In
+tracker-monitor::item-created: In
+tracker-monitor::item-updated: In
+tracker-monitor::item-attribute-updated: In
+tracker-monitor::item-deleted: In
+tracker-monitor::item-moved: In
+tracker-crawler::check-file: In
+tracker-crawler::check-directory: In
+tracker-crawler::check-directory-contents: In
+tracker-crawler::directory-crawled: In
+tracker-crawler::finished: In
+file-created(): Out
+file-updated(): Out
+file-deleted(): Out
+file-moved(): Out
+directory-started(): Out
+directory-finished(): Out
+finished(): Out
tracker-miner
+started(): Out
+stopped(): Out
+paused(): Out
+resumed(): Out
tracker-miner-fs
+tracker-indexing-tree::directory-removed: In
+tracker-file-notifier::file-created: In
+tracker-file-notifier::file-updated: In
+tracker-file-notifier::file-deleted: In
+tracker-file-notifier::file-moved: In
+tracker-file-notifier::directory-started: In
+tracker-file-notifier::directory-finished: In
+tracker-file-notifier::finished: In
+process-file(): Out
+process-file-attribute(): Out
+ignore-next-update-file(): Out
+finished(): Out
Signal flow of Metadex
metadex-miner
+tracker-miner-fs::process-file: In
+tracker-miner-fs::finished: In
+tracker-indexing-tree::directory-added: In
+tracker-indexing-tree::directory-updated: In
+metadex-finished(): Out

Figure 3.1: A graph visualizing the signal flow, emitters, and receivers within `libtracker-miner`. Inheritance is indicated using arrows, and signal flow is indicated using the diamonds as arrow heads.

`TrackerMinerFS` eventually hands over control to the implementing class, which contains specific instructions on how to mine and extract data from the file on which the event was raised. There are two main examples of this in the Tracker project; the `Files` miner and the `Applications` miner.

3.1.3 Performance tuning of TrackerMinerFS

`TrackerMinerFS` has three different properties which can be configured to greatly impact performance. These properties were designed to be changed when implementing new miners. Some of the properties are configurable in the original Tracker miners, while others are not, and are simply set to static values. The sections below will describe the different properties, how they are used in Metadex, and give a description of their respective performance impacts.

processing-pool-ready-limit

This is a unsigned integer property that sets the maximum number of SPARQL queries that can be kept in the SPARQL buffer before the buffer is flushed to the back-end database. The buffer is flushed either when it is full, a timeout has occurred, or when the miner does not have more work to do. The default value is 1, which means each query is flushed to disk instantly. With the hardware used in this project, it is more beneficial to raise this value significantly, in order to fill the write buffers of the operating system and thereby reduce disk IO.

processing-pool-wait-limit

This is a unsigned integer property that sets how many files can be kept in the internal task pool before processing of these files is forced.

throttle

This is a property ranging from 0.0 to 1.0, which tells the miner if it should index at full speed or not, where zero is full speed and one is the slowest setting, which adds a delay of one second to the processing of each file. The delay is achieved by adding the processing function to the GLib main loop either with high priority or as a function which is called periodically.

3.2 Nepomuk-KDE

Nepomuk-KDE is developed using technologies which align very well with the in-house competence of Pelagicore, and this is the main motivation for considering it as a base for Metadex. As previously hinted in this chapter (and better motivated in chapter 4), Nepomuk-KDE was not used in this project, but it is still interesting to observe the similarities in the architecture between Nepomuk-KDE and Tracker.

3.2.1 Overview of applications

Nepomuk-KDE is architecturally similar to Tracker (see section 3.1.1), it is designed in a modular manner, where each process has a specific responsibility. For inter-process communication D-Bus is used. The core processes are described below:

- `nepomukserver` is used to bootstrap the other Nepomuk-KDE processes, it is responsible for spawning different subsystems such as storage, file indexers and file watchers. The name itself is, according to the Nepomuk-KDE developers a bit misleading, and it should not be considered to be a server since none of the other services connects to it, or try to communicate with it.

- `nepomukservicestub` is a wrapper process for Nepomuk-KDE services. It is used by `nepomukserver` which spawns it with the appropriate service name as argument. Nepomuk-KDE ships with three basic services:

  - `nepomukfileindexer` which is responsible for scheduling and deciding which files to be indexed on the system. It relies on `nepomukindexer` to extract metadata from the files. `nepomukindexer` uses a series of libraries for extracting metadata from files, for instance, *FFmpeg*[8] and *taglib*[9] are used for audio and video, *Exiv2*[10] is used for DSC files and *poppler*[11] is used for PDF files.
  - `nepomukfilewatch` is responsible for listening for `inotify` signals from the kernel about file movement, deletion, and creation. This is the same notification system that is used in the Tracker `FileNotifier` module. Upon receiving an `inotify` signal, `nepomukfilewatch` updates the database via `nepomukstorage` and calls the file indexer for possible re-indexing.
  - `nepomukstorage` is the process responsible for accepting requests to the database over D-Bus. When a request is received by `nepomukstorage` it is passed to *Soprano*[12], which is a Qt framework for RDF data. Soprano is also used for parsing SPARQL queries to the appropriate format for the database back end, which in this project was the *Virtuoso Universal Server* from OpenLink Software[13].

[8]See `http://www.ffmpeg.org/`
[9]See `http://taglib.github.io/`
[10]See `http://www.exiv2.org/`
[11]See `http://poppler.freedesktop.org/`
[12]See the Soprano project page for more information: `http://soprano.sourceforge.net/`
[13]See `http://virtuoso.openlinksw.com/`

4

Benchmarks

In this chapter the design of the benchmarks, and the selection of benchmarking criteria is outlined. The purpose of these benchmarks is to compare the Tracker and Nepomuk-KDE metadata miners. The benchmarks are used to motivate the choice of which of these two systems to base the Metadex software upon.

4.1 Benchmark descriptions

The main criteria of our benchmarks follow naturally from the final requirements of the finished software developed from this thesis. Since a low mining time is crucial, the time spent on mining by each application was measured, discarding time required for tasks such as start up and initialization of databases. Both programs are typically kept running as background processes in order to receive monitor events from the operating system.

The system resource consumption levels must also be kept as low as possible. Both Tracker and Nepomuk-KDE persist data similarly, and each produce a database of the metadata of the indexed media. The database is stored as a file in a file system. The file size of the database produced by each indexer was measured. Since the software will run in a system with limited storage, the database must be kept as small as possible. The graph in figure 4.3 shows the file size of the databases for varying number of indexed multimedia files.

The benchmarks should not be seen as definitive, but rather as a simple model of the scenarios in which the indexers would be used in our environment. Both Nepomuk-KDE and Tracker were tested without modifications.

4.2 Selection of test data

The data used for the benchmarks was chosen to reflect the most common use cases of the software. Since the software is used to index multimedia, a range of multimedia items were gathered from the internet. *Wikimedia Commons*[1] and *Wikivoyage*[2] both have a large amount of high quality (in terms of available metadata) photographs and pictures which are free to use. These two sites were our main sources for pictures and photographs.

For video files, movie trailers were used to build a database of videos with high quality metadata. These trailers are smaller in file size than the typical size of a full feature film, but the metadata should represent the actual film very well. Due to the typical file size of a full feature film, the amount of trailers gathered is much lower than the number of pictures gathered, in order to more correctly reflect a typical use case (it is probably not common to have 20000 films on a nomadic device in a car, while this number of photos is not improbable). The *Internet Archive*[3] also supplies a high number of video files with varying metadata qualities, these files were used to represent items which may not contain perfect metadata.

Finally, for music, the idea is much the same as for video. A large quantity of music licensed under the *Creative Commons*[4] license was gathered from various online sources, and was used to model a typical music database.

The gathered data is used as a source for the generation of a realistic multimedia library, such as a removable USB mass storage device described in section 4.3.

4.3 Automatic generation of a media library structure

In order to simulate a real multimedia library, which can have an arbitrary directory structure, and contain an arbitrary selection of files, a media library is generated prior to the beginning of the benchmark. In each data point, additional files are added, which means the data from the previous point is always included in subsequent points.

The multimedia library generator selects files from the collection of gathered multimedia files, and generates a arbitrary directory structure to place these files in. An example of a generated media library tree can be found in figure 4.1.

The most important properties of the randomly generated file and directory structure are that the number of files increases from low numbers to high numbers as the benchmarks progress, and also that the file types included in the directories are mixed. By increasing the number of files after a set amount of benchmark iterations, the performance properties of each multimedia indexer can be measured for all media collection sizes.

[1]See `http://commons.wikimedia.org`
[2]See `http://www.wikivoyage.org/`
[3]See `http://archive.org`
[4]See `http://creativecommons.org`

Media type	Number of files	Total file size
Music	3078	35753 Mb
MP3	2507	28140 Mb
Ogg	560	7580 Mb
WMA	11	33 Mb
Video	452	24250 Mb
MP4	370	18850 Mb
Ogv	82	5400 Mb
Pictures	22927	13180 Mb
JPEG	16847	9460 Mb
PNG	4051	390 Mb
SVG	1619	600 Mb
GIF	410	30 Mb
Totals	25914	71 Gb

Table 4.1: Summary of the types of media used when benchmarking Metadex

```
data/
|-- gzgqnwxxzc
|   '-- zuzqxr
|       |-- kpx
|       |   |-- Alexandria_2123021.jpg
|       |   |-- AlexBiblDistance.wmv
|       |   |-- Orologio.jpeg
|       |   '-- Rabat_tour_Hassan.jpg
|       |-- PeitlerkofelgeologBruch.jpg
|       |-- Thron.jpg
|       '-- TihnaGebelUnknTombOutside.mov
'-- xcbnyw
    '-- cyy
        |-- LocationMauritania.png
        |-- Nafplio_große_Moschee.gif
        '-- Nz_Abel_Tasman_NP_Adele_Island.mp3
```

Figure 4.1: A typical generated directory structure for a media library

4.4 Structure of the benchmarks

Different methods are used for measuring the mining time required by Tracker and Nepomuk-KDE. For Tracker, the textual output of the miner process is enough to produce accurate statistics, while there is no such output from the Nepomuk-KDE process. The control and monitoring of Nepomuk-KDE is performed using D-Bus commands. The directories to index by Nepomuk-KDE are set using the `indexFile` function of `org-.kde.nepomuk.services.nepomukfileindexer`. The D-Bus signals `indexingStarted` and `indexingStopped` are monitored for activity in order to decide the time required for indexing.

For each of the different media library sizes, the actual benchmarking procedures are run five times. The purpose of running the same tests multiple times is to reduce the influence of potential outliers and unfavorable scheduling by the operating system. The arithmetic mean of gathered run times is computed from data stored by the benchmarks run previously.

In order to ensure that prior computations do not influence future benchmarks the operating system caches for directory entries, inodes and page caches are cleared using the `echo 3 > /proc/sys/vm/drop_caches` command. Prior to running `drop_caches`, two calls to `sync` are also made, in order to flush the file system buffers. Before starting the media indexers, each respective database is cleared, and this incurs additional overhead when starting the media indexers during benchmarking, which is why this delay is not included in the actual time calculations. In actual deployment of the software system the databases should not require resetting.

Since the benchmarks are meant to give a rough guide in which system we should base the rest of the project on, and not to be a definitive performance guide, we have run these benchmarks on standard PC hardware, and not on the actual target platforms. The main purpose of these benchmarks is to compare Tracker and Nepomuk-KDE, and not to focus on the run times in seconds, emphasis should be placed on the differences between the curves in the graphs, as well as the rates of increase in the graphs.

Nepomuk-KDE frequently stopped processing files during the running of these benchmarks, and thus had to be monitored for inactivity and subsequently terminated by an external program. This made the Nepomuk-KDE benchmarks difficult to run, and it is possible that the numbers for Nepomuk-KDE would have been more favorable if the process had functioned as intended. In addition to stopping sporadically, Nepomuk-KDE also took a long time to process large numbers of files, and the benchmarks had to be terminated when Nepomuk-KDE took more than 20 minutes for a single benchmark iteration. The time limit of 20 minutes was set rather arbitrarily, a time limit was needed in order be able to continue with the next benchmark iteration since it was not possible to decide if Nepomuk was still processing or had stalled.

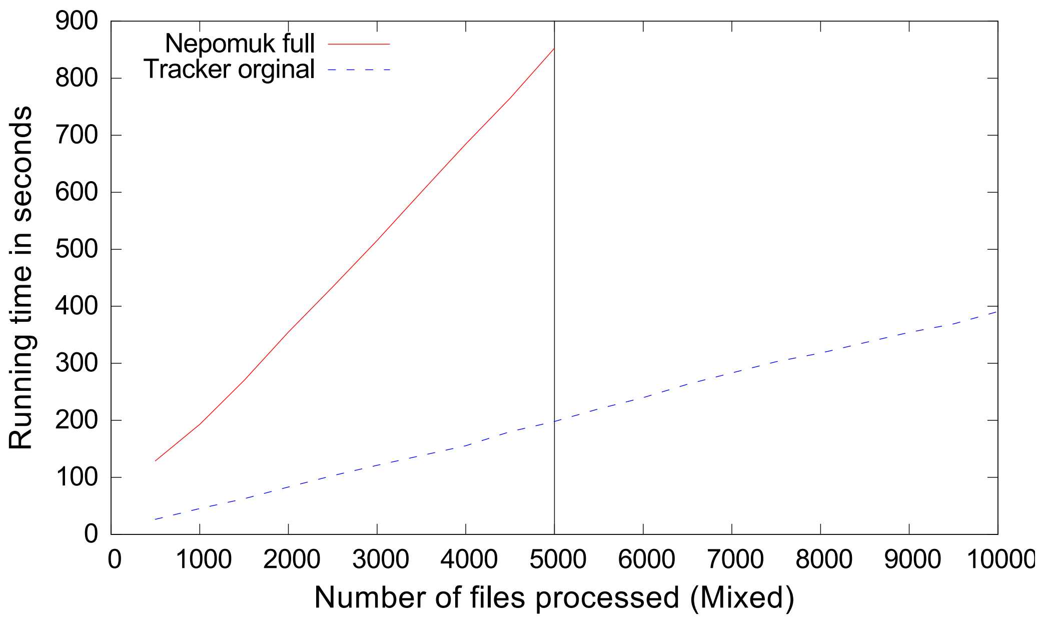


Figure 4.2: The running times of Nepomuk-KDE and Tracker. Tracker is clearly faster than Nepomuk-KDE, and also has a lower rate of increase in running time as the number of files increases. After 5000 files Nepomuk-KDE takes more than 20 minutes to complete, and is terminated.

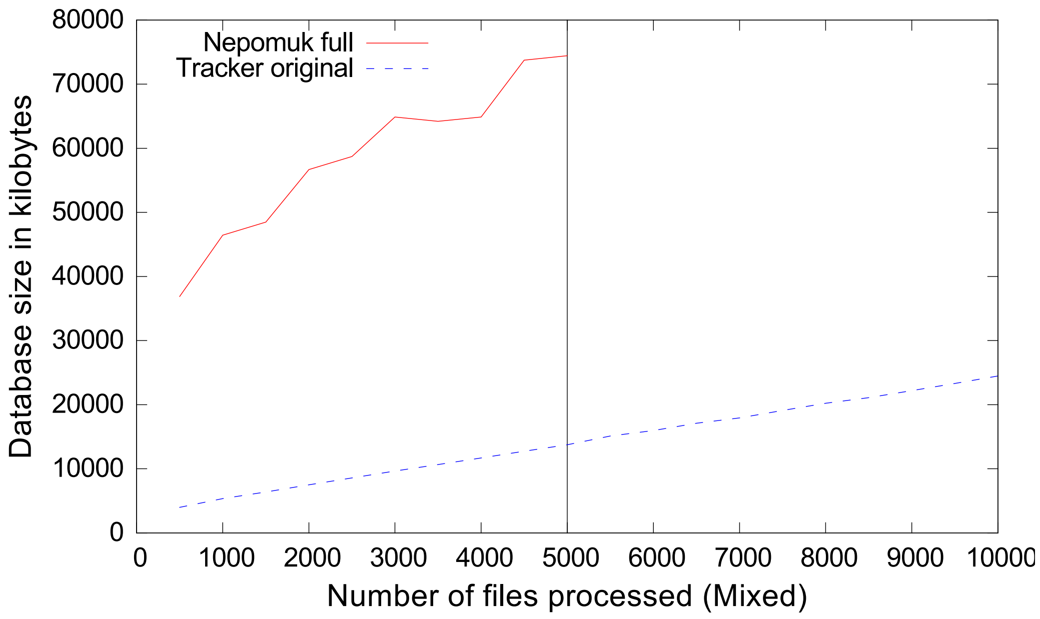


Figure 4.3: The size of the databases generated by Nepomuk and Tracker respectively. After 5000 files Nepomuk-KDE takes more than 20 minutes to complete, and is terminated.

4.5 Conclusion of the benchmarks

The benchmarks show that the Tracker system is faster and more efficient than the Nepomuk-KDE system. In terms of indexing time the Tracker system outperforms Nepomuk by several hundred seconds, as can be seen in figure 4.2. Also seen in 4.3, Nepomuk-KDE consumes an unpredictable and large amount of disk space for storing its database. In addition to consuming much more time than Tracker when indexing and mining, and using more disk space, the Nepomuk-KDE indexer failed to complete during several runs. While it is possible to terminate or restart the process using an external software, there are no performance benefits in using Nepomuk-KDE over Tracker and thus there are no reasons to use Nepomuk-KDE besides it being developed in the familiar Qt framework, which is not sufficient reason to choose Nepomuk-KDE over Tracker.

The dependencies of the Nepomuk-KDE system on the KDE libraries pose too much overhead to be included in the final product, and the removal of these dependencies seems time consuming. The lack of dependencies in the Tracker system makes it favorable in comparison to Nepomuk-KDE.

Based on the outcomes of the evaluation of the two candidate software packages, Tracker appears to be the most suitable software for our purposes, and the final product was therefore based on Tracker.

5

Implementation

This chapter describes the implementation of Metadex. The intention of this chapter is to document the modifications and additions made to the Tracker system, and the motivations behind these modifications and additions. In order to give the reader a self-contained description of the changes made to various modules, a background to the modified Tracker module is also given in this chapter.

The first sections of this chapter outline the changes made to various Tracker components, such as the `Crawler` and the `Store`. Later sections describe new, or heavily modified components, such as the Metadex miner and components used for configuring Metadex.

5.1 Overview of the miner

The diagram shown in figure 5.1 is used to show how the different modules of the miner interact. The three squares of the figure each denote both a logical module and an actual separate file in the code base, the *disk* graphic represents a collection of modules necessary for interacting with the disk, the specifics of these are however not very interesting. The *miner core* graphic also represents a collection of modules considered the core of the mining system (whereas the other modules in the figure could actually be used for other purposes as well).

The numbers by the arrows in the graph indicate the order of the control flow within the system. The miner core, which in Tracker is the `TrackerMinerFiles` object, and in Metadex is the `MetadexMiner` object, is used to start the rest of the modules. During initialization, the `IndexingTree` is populated with the directories to be indexed (①), as well as a parameter indicating whether indexing is to be recursive, and any file patterns to omit.

The `FileNotifier` is notified of the changes to the `IndexingTree` (②), and triggers activation of the `Crawler` (③) for each of the directories in the `IndexingTree` (④), and

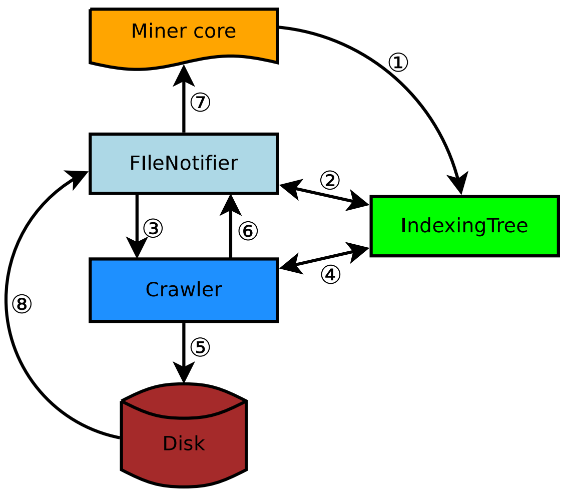


Figure 5.1: A simplified diagram over the different components of a miner

the `Crawler` fetches information from the file system (⑤). When a directory has been fully crawled, the `FileNotifier` is alerted (⑥), and eventually triggers the processing of the crawled files in the miner core module (⑦). Finally, changes to files on the disk can directly trigger events in the `FileNotifier`, which are also sent to the miner core for processing (⑧).

The sequence diagram in figure 5.2 shows in greater detail the flow of signals / method calls when executing the Metadex binary file, until it receives the `metadex-finish` signal, when fully mining a single file from the file system. It also includes the initialization of the `Configuration-`, `Removable media-`, and the `FileIndexer` object, which does not have anything in common with just mining a file, but is necessary for the additional features of Metadex. This graph could help the reader to understand the dependencies between objects and the communication flow between the different objects and the external processes. The following sections describe the different modules mentioned in greater detail.

5.2 Crawler

The crawler is responsible for finding all files in the directories specified by the `IndexingTree` module. The directories specified by the `IndexingTree` are called *root directories*, since these directories specify the topmost directory where all child directories and files should be indexed. The crawling takes place by simply iterating through the directory structure breadth first. Once a file, or directory, has been found, several signals may be emitted. The following list of signals is for the unmodified crawler, and in the following section the modifications made to it will be presented.

- `check-directory` is used to ask the `FileNotifier` whether a specific directory should be crawled or not. The `FileNotifier` makes this decision by checking

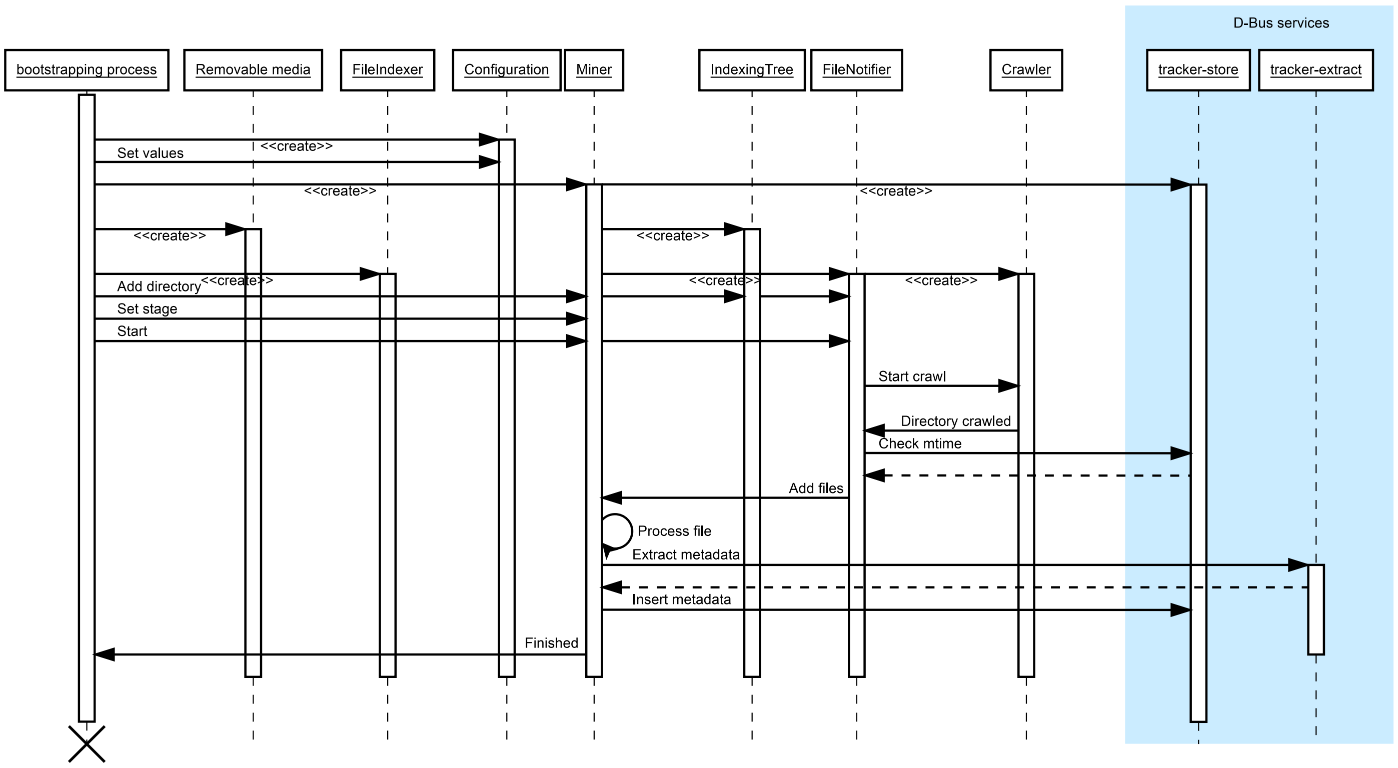


Figure 5.2: Metadex sequence diagram over the mining of a single file, in the non-daemonized mode. Solid arrows indicate function calls, dashed arrows indicate function return values. Unlabeled arrows should be seen as continuations of the arrow leading in from the left.

if the directory is a child of a directory specified for recursive indexing in the `IndexingTree`, and various filtering settings in the `IndexingTree`. If the response to this signal is `True`, the directory will be crawled. This is called once per directory during the crawling process.

- `check-file` works in the same way as the `check-directory` signal, but is used to decide whether a file should be indexed. This is called once per file during the crawling process. If the file for which this signal is emitted does not match the `allowed-file-patterns` option the file will be omitted.

- `directory-crawled` is used to notify the `FileNotifier` that a root directory has been crawled and is ready for further processing. This is emitted once per root directory.

In the original implementation of the `Crawler`, a root directory is picked from the `IndexingTree`, each file and directory contained in the root directory is crawled until all files in the current root directory have been visited. Before picking a new root directory, control is passed over to the `FileNotifier` module, which in turn registers the status of the recently crawled files (more on this in section 5.3). When the `FileNotifier` has finished registering the files, it triggers the crawling of a new root directory in the `Crawler` until there are no more roots to process. In 5.1 the arrows 2-6 show the communication between the `FileNotifier`, `Crawler`, `IndexingTree` and the `Disk`.

5.2.1 Short-circuited file crawling

It is sometimes useful to be able to present arbitrary media to a user, such as when a new device is discovered and the user should be notified that the media on the device is available. When no specific search query is given to the system, and any media metadata will suffice as a response it is thus more important to respond quickly than accurately.

In order to quickly respond to these unspecific queries, Metadex can be configured to quickly insert the first n files crawled regardless of their content. This quick insertion is called *short-circuiting* in Metadex. The number of files to be short-circuited can be configured by the user of the system.

If short-circuiting is enabled, the crawler will record the number of files crawled, and when n files have been crawled, crawling will stop and the `directory-crawled` signal is emitted to the `FileNotifier`. In Metadex, the `directory-crawled` signal has been modified to accommodate a status of the short-circuiting, i.e whether it is currently active or not. The `directory-crawled` signal is normally only emitted once per root directory in the unmodified Tracker crawler, but in Metadex this signal may thus be emitted several times per directory. When the files gathered during the short-circuit have been processed by the miner, the crawler is started again and continues crawling where it left off.

The `Crawler` object is initialized by the `FileNotifier` constructor in the original Tracker implementation. In order to pass the new attributes for short-circuiting to the `Crawler`, the constructor of the `Crawler` was modified. Due to the design of constructors in GObject (see section 2.7.1), the parameter values for the `Crawler` cannot be passed to it via the constructor of `FileNotifier`, but has to be set using a special initialization function, called manually when creating the `FileNotifier`.

5.3 FileNotifier

The `FileNotifier` is used to keep track of events occurring to files before these events have been processed by the miner, this module can be seen as an advanced processing queue for the miner.

When the crawling process has finished (regardless of short-circuiting), the `File-Notifier` processes each file in order to determine to which of the internal queues the file should be added for further processing. There are three possible internal queues in the `FileNotifier`; the queue for deleted files, created files and updated files. The `FileNotifier` uses the stored modification times, and compares these to the current modification times in order to decide which queue the file should be put in. Later when files are processed by the miner, the miner has direct access to queues created by the

FileNotifier.

In Metadex, the parameters for the Crawler must be passed through the File-Notifier, since the Crawler is initialized via the FileNotifier. The FileNotifier can use the value intended for the Crawler in order to decide when the system is running in a short-circuited mode, but it is impossible for the FileNotifier to use this value to decide the current internal state of the Crawler. The Crawler can be in two states when short-circuiting is enabled; either short-circuiting has previously been completed, or it has not been completed and will be completed after the current run.

When crawling has been short-circuited, only the n first files will be sent to the File-Notifier, and thus the other files, not yet crawled, will appear to be deleted since their modification times are present in the store, but not detected by the crawler (assuming this is not the very first run of the crawler). In order to mitigate this, the File-Notifier must be notified when a run of the crawler has been short-circuited, and this information is passed along from the Crawler to the FileNotifier using a modification to the directory-crawled signal.

5.4 The Store module

The Store module is used to communicate with the database, which in this project is SQLite. Queries are sent to the Store in SPARQL format, and must be translated from SPARQL to SQL before being processed by the database. The translation from SPARQL to SQL is performed by the libtracker-sparql library. The Store module is largely unchanged, with the exception of the added capability to monitor changes in query results, described in section 5.4.1.

5.4.1 Query monitoring

Due to the design of Metadex, where the user is notified of query hits as quickly as possible, and the query result is designed for speed and not accuracy in the first stage of indexing, it is important to receive notifications when further metadata has been analyzed. Metadex allows users to *monitor queries* for this reason, a monitored query looks the same as a regular search query (and is handled in a similar way by the system), but is run several times and the querist is notified of changes in the query results.

Metadex allows users to specify a SPARQL query and an interval in milliseconds at which to run the query against the database. When presented with a monitoring request, Metadex responds with a query identification number which will be used to tag D-Bus broadcast signals with new results. When a new result to a query is found, a D-Bus signal is sent, and any interested party can subscribe to these signals and react appropriately when the signals are received. The same identification number is used to unsubscribe from, and cancel the signal.

5.5 Introspection-based prototypes

A main reason to base the project on the existing Tracker software was to shorten the time required for implementing the required features. In order to further shorten development time, the first prototypes of Metadex were implemented using *GObject introspection*, in Python and Vala. If the prototypes were successful, the entire project would be based on these introspection technologies.

Most of the re-usable parts of the Tracker software are implemented as GObject C libraries, and the purpose of GObject introspection is to enable applications written in languages other than C to use GObject (see section 2.7.1) C libraries. The PyGObject[1] library allows Python programs to use GObject libraries, by reading the metadata generated from the C source files of the GObject library. PyGObject dynamically creates wrapping code around the native GObject code and allows Python to call functions, inherit classes, etc. as if the entire code base was written in Python.

Vala works in a similar way as Python in this respect. Vala uses *vapi* (files specifying the interfaces to libraries) files in order to correctly interface with the GObject libraries. The vapi files can, just like the metadata files used by PyGObject, be generated at compile time. While the Python code is run by the Python interpreter, the Vala code is compiled to C, and the expressions regarding GObject are expanded into proper C, much like if the code had been written in C and GObject, by hand.

The Python and Vala prototypes were implemented in parallel. Since the introspection technologies are so similar in both approaches, the actual interfacing with the Tracker libraries was expected to work in the same way in both prototypes, and the available functionality from the Tracker libraries was also expected to be identical for both prototypes.

The reason for creating two prototypes with the same features was to see which implementation would yield higher performance when speed was measured. Python is an interpreted language, and was believed to run slower than the Vala counter part, which essentially is a GObject C program with different syntax.

Basic interfacing with the Tracker libraries worked fairly well in both the Python and Vala prototypes, and surprisingly they appeared to perform equally well in terms of speed, however several problems made both approaches unusable for further implementation:

- Not all features of GObject are implemented in PyGObject. The main issue encountered here was the inability to connect to some signals emitted by the Tracker libraries

- The vapi files used in Vala to interface with the Tracker libraries are not automatically generated, which means they would need to first be updated to correctly reflect the current state of Tracker, and then be updated for each change made to Tracker in the future.

[1]see: `https://live.gnome.org/PyGObject`

- Modifications to the libraries became necessary early on in the development of these prototypes, which meant a C code base and also a Python/Vala code base would need to be maintained, rather than just C or just Python/Vala.

The reasons stated above motivated the development of a new prototype using GObject and C, in order to mitigate the issues with introspection. The rest of this chapter will describe features implemented in the C/GObject prototype (simply referred to as 'Metadex'), as well as modifications made to the Tracker libraries.

5.6 The Metadex miner

There are several responsibilities for the Metadex miner. The miner is, besides the Store, the main application run by the user of the system, and is communicated with over D-Bus. The Metadex miner binary replaces the `tracker-miner-fs` binary supplied by Tracker.

The miner bootstrapping process is responsible for initializing the different modules of Metadex. The bootstrapping process starts by sanitizing the program arguments, initializing the configuration object with the arguments, setting up the logging environment, and starting the mining process.

The miner process is entirely event driven. Once the mining process has been started it waits for events generated by the `FileNotifier`, which in turn triggers the `process_file` function of the miner, described in section 5.6.1.

5.6.1 Mining in multiple stages

One of the hypotheses of this project is that the time required to produce a first result to a search query is lowered if the metadata is mined in multiple stages, where the first stages provide less detailed metadata than the later stages. There is no support in Tracker to extract metadata in this way, and therefore several subsystems in Tracker were modified to support this.

The different stages of Metadex can be seen as different miners, since the Metadex program can be instructed to run only one of them. The stages are however kept in the same module, which allows either of the stages to be executed at any time during the running of Metadex, without needing to spawn a new miner process.

The ability to switch stages during Mining, i.e. processing a file fully using the second mining stage even if the Metadex system was configured to only run in the first stage, is useful when receiving queries via the `FileIndexer` module (see section 5.6.2). `process_file`, being the entry point in to the miner from the `FileNotifier`, is used as a divider, deciding which action to take for each file.

There are three distinctly different classes of `GFile` objects processed by `process-_file`; directories, files to be processed in the first stage, and files to be processed in the second stage of mining.

Metadata properties of the first stage

The first stage of mining extracts metadata properties which are *quickly accessible*. Whether a property is quickly accessible is obviously a subjective matter, and a balance between the usefulness of the metadata and the time required to access the metadata must be found. While the time needed for the actual extraction of the properties must be low, the number of properties (regardless of accessibility) must also be kept as low as possible since query creation and insertion also takes time.

For the first stage of extraction only attributes provided by the GLib `GIO` library are used. These attributes are fetched in to Tracker via `GFileInfo` objects, which are basically caches for the `getxargs` and `lstat` system calls, when Tracker is run on Linux. The reason behind restricting the first stage to the aforementioned properties is that the information retrieved by these system calls should be faster to extract than metadata which requires the file to actually be read. In table 5.1 each property extracted in the first stage of mining can be seen.

The properties mined are used to build a SPARQL query that inserts the mined data into the database.

Metadata properties of the second stage

The second stage of mining adds more properties to the already existing properties added in the first stage. The actual set of properties depends on the type of the file being mined. The insertion query of the second stage, containing the metadata, is partially generated by the extractor process and is received via a response from a D-Bus call to this process. There are many possible attributes, and they will not be listed here. It should be noted that the second stage of mining is drastically slower than the first stage. The slowdown comes from the following:

- much more data is inserted, an MP3 file may have up to 42 extra properties after analysis in the second stage of mining;
- the extraction service used during the second stage of mining is invoked over D-Bus, while the first stage of mining only uses local function calls;
- for some file types external processes are executed by the extractor service, such as `MPlayer`, `Xine` or `GStreamer`, and this further adds to the mining time;
- for some file types there are extractor libraries available (and hence no new process needs to be started as in the previous point) - while this is faster than using an external process it still takes extra time compared to the first stage of indexing.

5.6.2 The FileIndexer module

Normally the `Crawler` process adds all files to the internal processing queues. It is sometimes useful to add files to the processing queues even though they have not gone

	Attribute	Purpose
Full URL	`nie:url`	Full URL to the file, this is known since the file was reached by the crawler when traversing the directory structure, and does not need to be retrieved using any extra calls. The URL is formatted to conform to the standards used in Tracker.
File name	`nfo:fileName`	The actual name of the file, does not include the path. As above, this is known from crawling
MIME type	`nie:mimeType`	This is guessed from the file extension of the file, which makes it much faster than using the register of known MIME types typically available from the operating system
Title	`nie:title`	This property has different meanings depending on the type of media being mined, however, in this stage, this is always set to the same value as `nfo:fileName`.
File size	`nfo:fileSize`	The size of the file on disk, this information is retrieved from the cached `lstat` value of a `GFileInfo` object
Last modified	`nfo:fileLastModified`	The time of last modification, also retrieved `lstat` from the cached `lstat` value of a `GFileInfo` object
Last accessed	`nfo:fileLastAccessed`	The time of last access, also retrieved `lstat` from the cached `lstat` value of a `GFileInfo` object

Table 5.1: Table over attributes mined during the first mining stage

through the crawling process, in Metadex a use case for this feature is requesting metadata for files through the user interface of a client application, such as the media player of the IVI platform. Communication with this module takes place over D-Bus, and any application can request a file for processing.

In the original implementation of the `FileIndexer`, the files were simply enqueued, and any ongoing crawling or mining would precede the processing of the files added by `FileIndexer`. In Metadex, files added by the `FileIndexer` are always processed first, the current indexing stage is disregarded, and a complete metadata extraction is always made.

The Metadex miner keeps a hash table of all files added via the `FileIndexer` module, since this module itself is not aware of the different stages of mining. When a file is added via the `FileIndexer`, Metadex stops what it is currently doing, adds this file to the front of the processing queues and pushes it to the miner, which checks the hash table to decide if this file should be treated specially.

The hash table lookup is used by the dividing functionality in `process_file` mentioned in section 5.6.1. `process_file` is sent a `GFile` object, and the URI of this object is compared to the keys of the hash table, if there is a match in the hash table, the file of this URI should be processed fully regardless of the current stage of Metadex.

5.6.3 Configuration

Tracker uses a configuration module as a proxy between the configuration system of the operating system (typically `dconf` and `gsettings` in Linux) and the internal properties of Tracker. The configuration object can be used to, for instance, change the list of directories to be indexed during run time, and have Tracker react to these changes as they occur in the operating system.

In Metadex, it is not important to change configuration properties during run time, since these properties will be set when the application is executed. Typically the users of Metadex will have limited configuration abilities, and the settings are decided during the construction of the software platform.

Metadex instead optionally reads configuration options from `gsettings`, but never writes these options back to `gsettings`. This means the configuration object can be used to make non-persistent choices, only valid for one run, and can be passed around to the different subsystems in Metadex without care for the persistent system settings.

Below is a list of the available configuration settings, these settings are changed using the `dconf` system, and reside in the `org.freedesktop.tracker.miner.metadex` namespace, and can also be set on the command line as parameters to Metadex.

- `allowed-file-patterns` is used to specify a list of glob patterns (a glob pattern is a simplified regular expression) which each file will be compared to, in order to decide whether the current file should be indexed or not. The inclusion of a wildcard pattern ('*') disables this feature and discards all patterns, thereby accepting all files for indexing.

- `index-recursive-directories` is used to specify a list of directories to crawl recursively. The directories specified here are put in the `IndexingTree`, and hold the files to be indexed. The command line parameter for this option is called `directories`.

- `index-single-directories` is used in the same way as the option above, however child directories are not crawled. The command line parameter for this option is called `single-directories`.

- `log-to-file` indicates whether the output sent to `stdout` or `stderr` should also be sent to a log file, typically storied in `/.local/share/tracker/`. The command line parameter for this option is also called `log-to-file`.

- `processing-pool-ready-limit` specifies the number of items kept in the internal buffer for SPARQL queries before the buffer is emptied and the items are written to persistent storage. This is discussed in section 3.1.3. The command line parameter for this option is also called `processing-pool-ready-limit`.

- `processing-pool-wait-limit` specifies the maximal number of files kept in the internal processing queues created by the `FileNotifier` before processing of these files is forced. This is discussed in section 3.1.3. The command line parameter for this option is also called `processing-pool-wait-limit`.

- `short-circuit-files` specifies the number of files to be processed in a short-circuited run before proceeding to process the remaining files, see section 5.2.1. The command line parameter for this option is also called `short-circuit-files`.

- `throttle` specifies the indexing speed. This is discussed in section 3.1.3. The command line parameter for this option is also called `throttle`.

- `verbosity` specifies the log verbosity for both file logging and logging to the console. The command line parameter for this option is also called `verbosity`.

6

Results

This chapter provides the numeric results gathered from Metadex, and where applicable, also measurements gathered from the Tracker software it improves upon. Different media types potentially incur different indexing times, and thus the measurements have been performed on both the different media types separately, and finally all media types combined, which is called ”mixed” media.

The data files used to produce these results are the same files used in the benchmarking section, and a discussion of these files can be found in section 4.2. Similarly, the directory structure holding the files is generated using the procedure described in the benchmarking section 4.3.

The tests were executed on a dual core work station, and thus the run times measured in seconds do not represent the actual run times of a (current) embedded system. The tests were run on the work station since they would take too long to run on a typical embedded system, due to the number of iterations for each test. The data of main interest from the graphs is thus the difference between the plotted data, and the slopes of the graphs, and not the measurements in seconds. It should be noted that the next generation of IVI systems are likely to feature dual core processors and several gigabytes of RAM, and these future systems are the main targets of Metadex.

Since Metadex is designed to run in a Linux environment, the Linux work station chosen to produce these test results should give a good idea of the performance of Metadex in a real IVI system. In order to reduce the influence of external factors in the system, unnecessary software was stopped before executing the tests. The operating system caches for inodes and dentries were flushed, as well as the cache for page entries. The work station was also periodically manually inspected using the `htop`[1] system monitor, in order to spot unfavorable resource allocation.

[1]See: `http://htop.sourceforge.net/`

6.1 Measurements and interpretations

The first graphs, seen in section 6.1, show the difference in speed between the two short-circuit modes (discussed in section 5.2.1). It can be seen that the short-circuited first stage has a stable and predictable run time. This stage does not rely on an external extractor program, and retrieves all information from the file system (rather than the actual file) and is thus mostly affected by IO latency and delays. The second stage short-circuited shows more fluctuations. There are several possible causes for these fluctuations; there can be problematic or broken files with corrupt metadata, causing problems for the metadata extractor process, the extractor process can be scheduled unfavorably by the operating system, or the communication between the miner and the extractor (the D-Bus connection) can be delayed for some reason.

This graph clearly shows differences in performance and predictability with the different extractor libraries used in Metadex (and in Tracker). The audio and video parsing libraries used in figures A and B both produce predictable results in comparison to the image extractor seen in C. These results clearly show that the image extractor libraries need further work. In the final figure, D, the same trends can be seen as in the figure of the image benchmark. The fluctuations in D are most likely due to the fluctuations in the image processing.

Since the runs are short-circuited, the running time of the miners does not noticeably increase as the number of files increases. The graphs of section 6.1 show the time required to produce the metadata for the short-circuited files only, and not the mining time for all files.

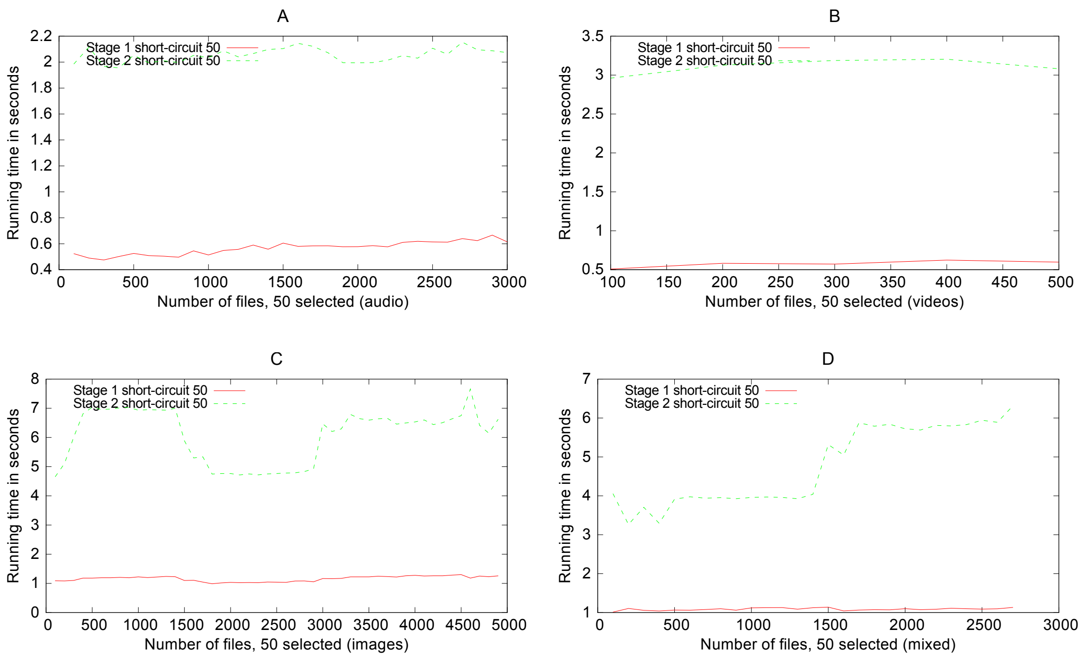
The next figure, figure 6.2, can be seen as a zoomed out version of the previous graph. In this version, all mining modes have been included. It can be seen that the short-circuited runs are, as expected, much faster than the full runs (albeit do not produce metadata for nearly as many files). Also seen in this figure is the difference in running time between the first and second stages. This difference is likely due to the same causes as for the first figure, where there is both a larger amount of work to be carried out in the second stage of mining, and there are also more external factors which can slow the second stage down.

In these graphs it can be seen that the same trends regarding the differences in reliability between the image, audio and video (figures A, B and C) extractors are still present in this larger scale run. The fluctuations in running time for images are not as striking, and this may indicate that a subset of malformed image files cause comparatively long extraction times. When reviewing the code for the image extractors, special attention should thus be given to the error handling procedures to see if there are potential speed improvements here.

It should be noted that the graphs of figure 6.2 do not depict the time required to produce search results. The reason for including both the short-circuited runs and the non-short-circuited runs are to give a perspective of how much faster the short circuited runs are than the full runs.

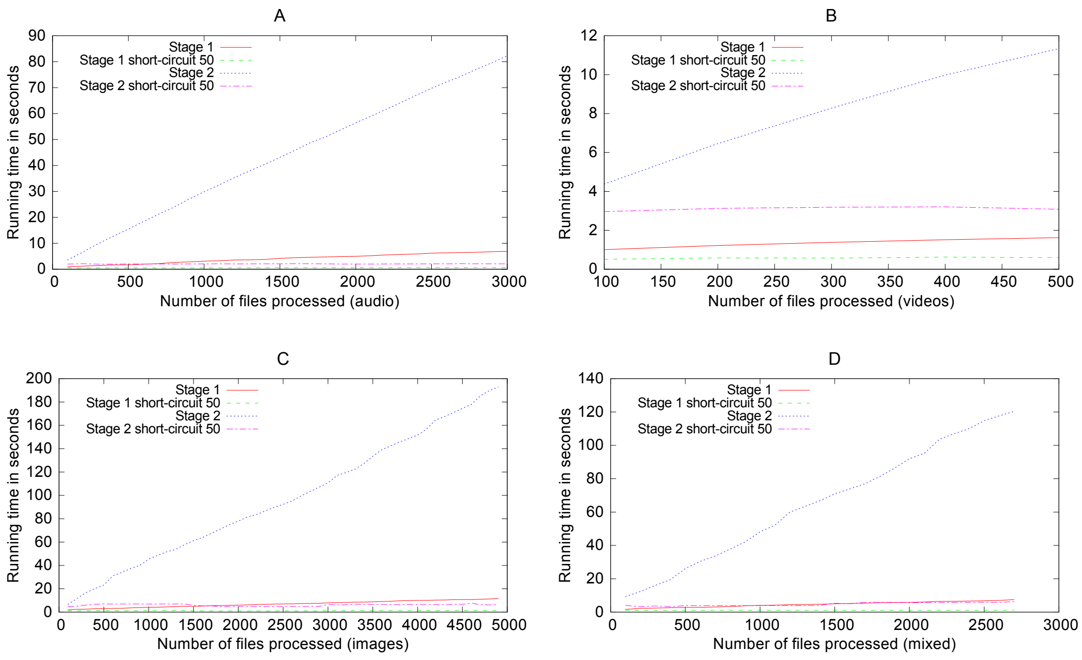
The graphs of 6.3 show the time required to decide whether a previously inserted

Figure 6.1: Metadex stage 1 and 2, short-circuited



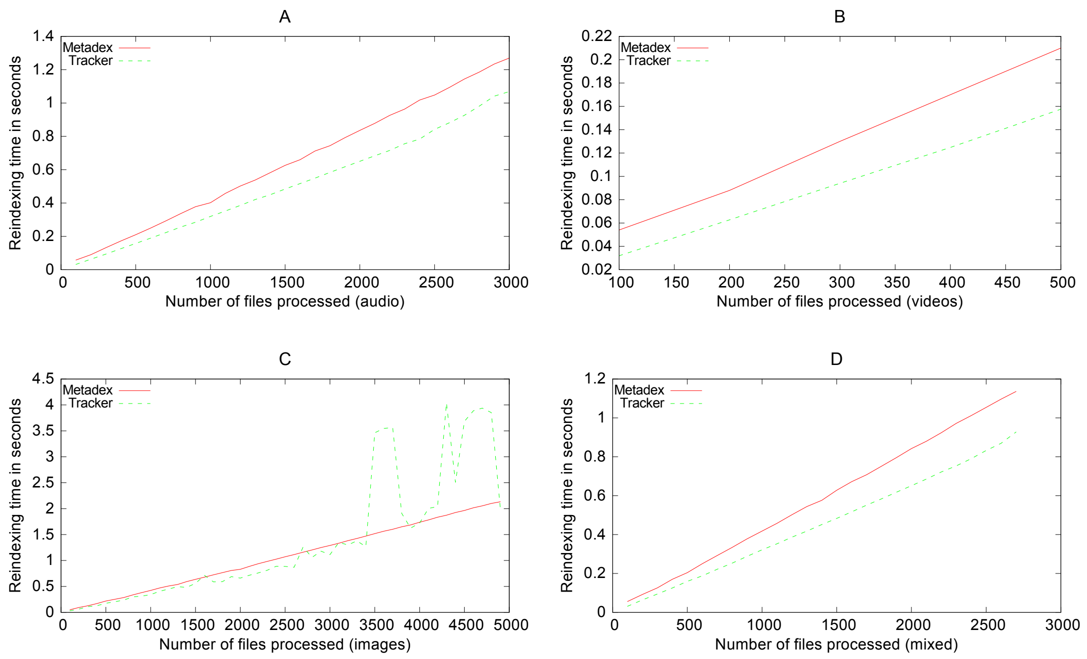
The time required for processing different kinds of media in the stage 1 and stage 2 short-circuit modes with Metadex

Figure 6.2: Metadex, all modes



The time required for processing different kinds of media, with all available Metadex configurations.

Figure 6.3: Re-indexing, Metadex and Tracker



The time required for re-indexing different kinds of media in Metadex and Tracker

storage device has already been mined. No files were changed between mining and the re-indexing, the graphs show only the time required to decide that no files were changed.

The graphs show that Metadex is both consistently slower than Tracker, and also has a steeper slope. Both Tracker and Metadex share the same procedures for crawling, where Metadex has some logic added for handling short-circuited runs. It is important to keep the re-indexing times low, so this result should be improved upon, possibly by changing the logic added to Metadex regarding the short-circuited runs.

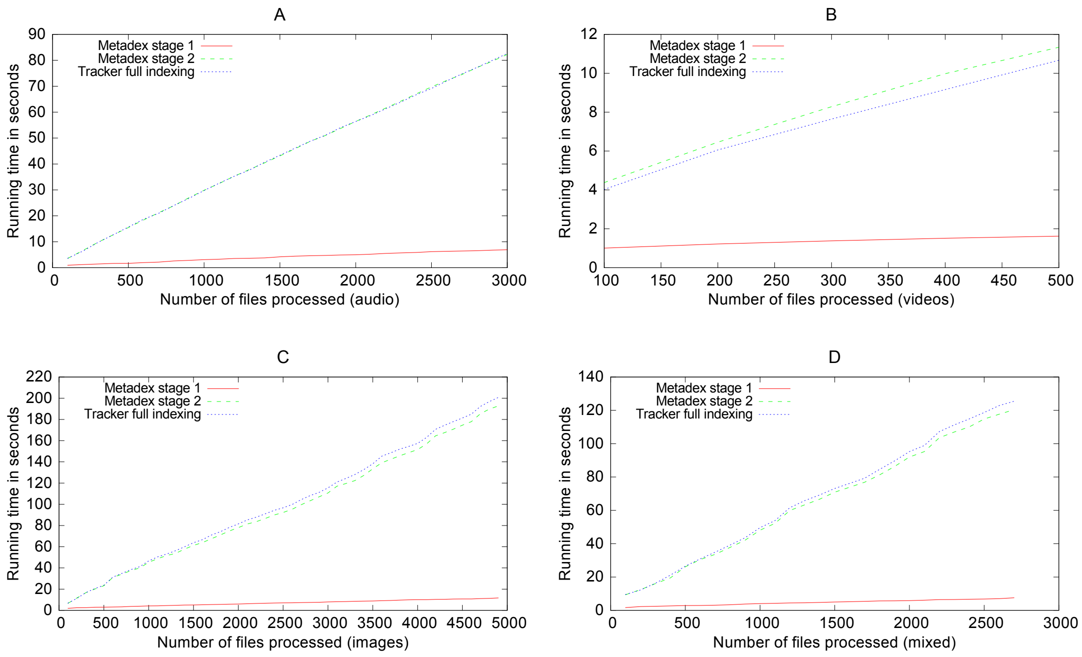
The graphs also show a higher consistency and predictability in the re-indexing times of Metadex than the corresponding times of Tracker. It is unclear why Tracker has some sudden increases in re-indexing time.

In figure C, several fluctuations in the re-indexing times for Tracker can be observed. These fluctuations are not present in the rest of the graphs, which may indicate that the fluctuations come from unfavorable scheduling by the operating system.

The graphs seen in figure6.4 display the running times for the non-short-circuited modes of Metadex, compared to a full mining of Tracker. These graphs can be used to establish the time required to process all files in the different stages, and when used in conjunction with, for instance 6.1, or 6.8, this gives a good impression of both the time required to retrieve first results, and subsequent arbitrary query results.

The graphs of figure 6.3, particularly B, show that the second stage of Metadex has a slightly longer running time than Tracker. This is likely due to the extra logic added in order to accommodate for the multiple stages of mining. This added overhead is

Figure 6.4: Runtimes, Tracker, Metadex stage 1 and stage 2, short-circuited



The time required for processing files using the two stages available in Metadex compared to the one stage available in Tracker

acceptable due to the low running time required by the first stage of mining.

The graphs in figure 6.5 show stage 1 of Metadex executed with short-circuiting enabled and Tracker which running a complete metadata extraction. These graphs give a perspective of the difference between the fastest possible configurations of Metadex, compared with a standard Tracker run. Since the running time of Tracker gets relatively large compared to stage 1 of Metadex and the short-circuited version of the first stage of Metadex, when the number of files increases, it looks as if the Metadex lines are constantly zero, however they are indeed greater than zero as can be seen in 6.1.

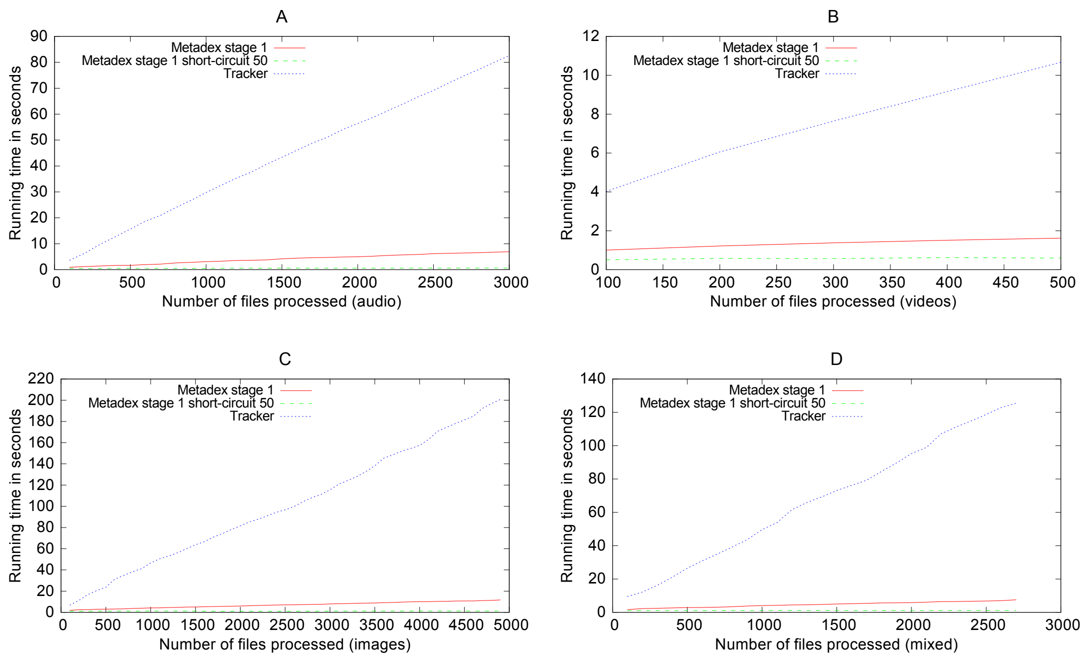
It should be noted again that Tracker produces all metadata available, and that in these graphs, Metadex only produces the most elementary metadata, and this is the reason for the drastically lower indexing time of Metadex in figure 6.5. The use case plotted in these graphs could be when deciding if a file is present in the system at all. In this scenario, Metadex will be able to decide the availability of a file much quicker than Tracker, due to only extracting the most basic metadata properties.

In figure 6.6 the running time in seconds for a varying number of directories has been plotted. Directory depth ranges from one sub directory to ten sub directories deep. The number of files was also kept constant, while the number of total directories was varied.

The hypothesis in this test was that the time consumed when running the different stages of Metadex and the Tracker miner would roughly be equal, as can be seen in the graph, this turned out to be false.

The slightly longer running time of Metadex stage 2 when compared to Tracker is

Figure 6.5: Metadex stage 1, stage 1 short-circuited, and Tracker



The best performance settings of Metadex (stage 1 mining, and short-circuited stage 1 mining) compared to the normal case of Tracker (full mining)

likely due to the modifications made to the crawler as was observed in figure 6.3.

Metadex stage 1 is much faster than both stage 2 and Tracker, this likely indicates that the extractors handle directory traversal inefficiently.

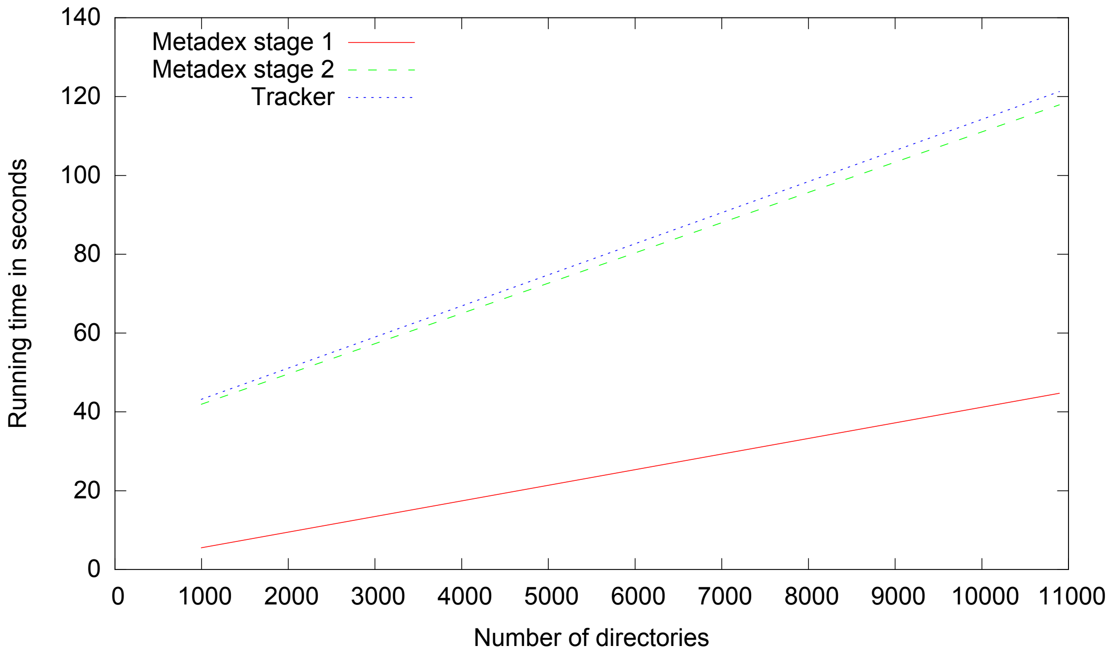
The graphs depicting the database sizes for Metadex and Tracker, seen in figure 6.7 are mostly intuitive. Both Tracker and Metadex stage 2 run in parallel and the first stage of Metadex constantly produces a smaller database than both Tracker and Metadex stage 2. It is however unexpected that Metadex stage 2 produces a smaller database than Tracker. This difference in size could be due to Metadex mining less data from each file, and thus yields less information when querying the database for a file.

The information available from both Metadex stage 2 and Tracker appears to be equal upon inspection, and thus it seems unlikely that Metadex stage 2 produces less information regarding each file, and more likely that Metadex stage 2 discards some other data which Tracker contains, not related to the mining of multimedia.

The time to first result, displayed in figure 6.8 shows the elapsed time between initiating a query against the store and receiving a response. Metadex features query monitoring capabilities, as discussed in 5.4.1, Tracker does not however, and for this reason the monitoring was not used when producing these graphs.

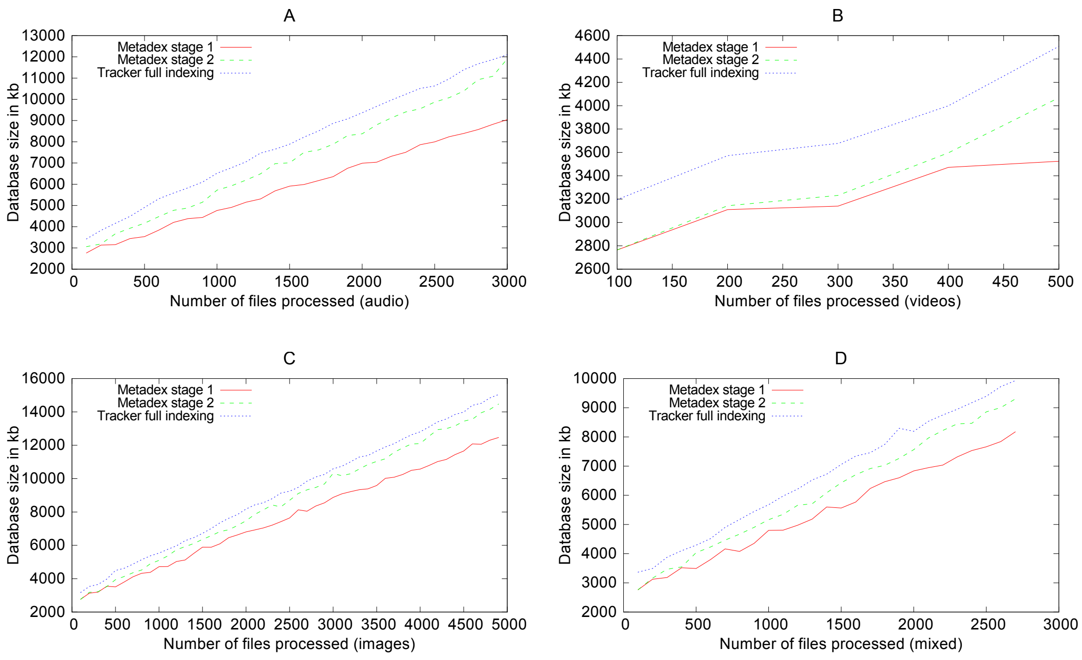
To produce these graphs, the `tracker-sparql` tool was used together with a SPARQL query which matches any file, of any type. The query tool is run every 100 milliseconds until the first hits are returned by the store. Since the tool is run periodically until a hit is returned, the database is hit more times than necessary, which means these readings

Figure 6.6: Runtimes, varying number of directories



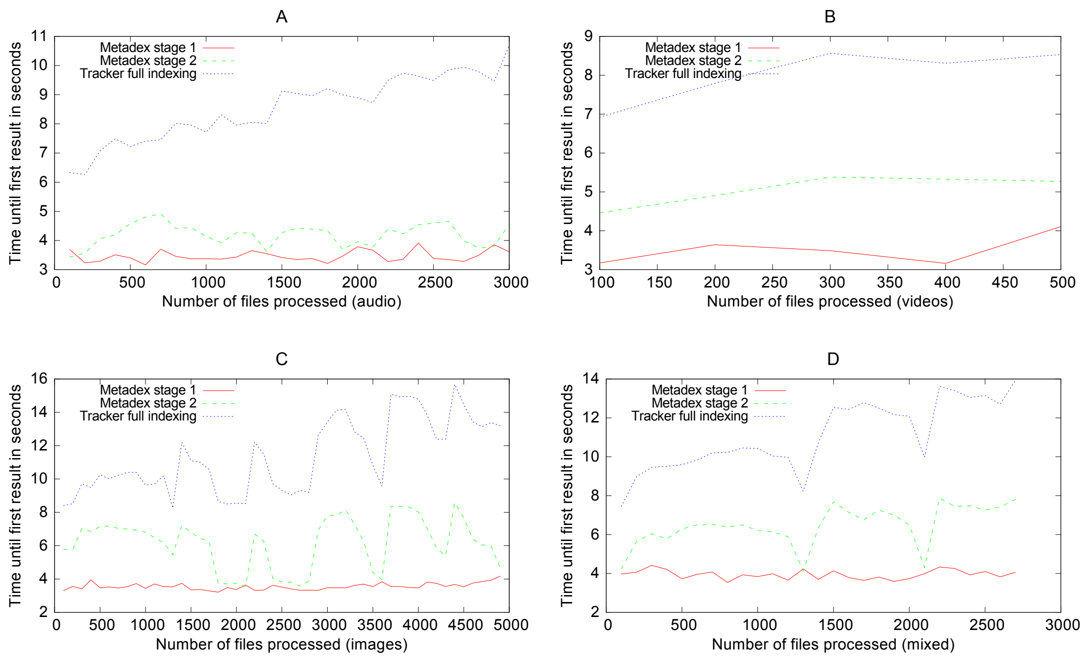
This graph shows how Metadex and Tracker are effected by large directory structures, while keeping the number of files fixed at 1000

Figure 6.7: Database sizes



Comparison of the size of the database files of Tracker and the various stages of Metadex

Figure 6.8: Time to first result, Metadex and Tracker



Time required to produce a first result by the Tracker and Metadex systems.

are not completely accurate, but the inaccuracies are the same for both Tracker and Metadex. It should also be noted that the 100 millisecond delay between the queries sent to the database means approximately 100 milliseconds may be added to the result measurement in the worst case.

Metadex is able to respond to queries for arbitrary files because of its short-circuiting capabilities. By comparing the graph of figure 6.8 to the previous graphs which have shown the running time of the short-circuited stage 1 Metadex miner (for instance 6.1), the time consumed by the actual querist and time time required by the database to locate the correct element can be approximated. The time required for Metadex to extract and insert the data for 50 arbitrary files (as seen in 6.1) is much lower than the time required to obtain a first result by a client, as depicted by these graphs, which could indicate that a large amount of time is spent in the querying client, and in the database server looking up the query results. As a side note, the first result times were considerably lowered, and the curves were smoother, when the operating system caches for pages, dentries and inodes were not cleared prior to each run.

Producing an early first result is one of the key goals for this project. The very first results are intended to be retrieved by running the first stage miner, and subsequently retrieving more information as gathered by the stage 2 miner, thus it is important to keep the time required to produce a first result low especially for the first stage of mining.

The relative stability of the stage 1 curve comes from the fact that the extractor process is not invoked, as mentioned previously. The close resemblance of the second stage of mining and Tracker can also be seen in that these curves almost run in parallel.

7

Conclusion and recommendations

This chapter describes the conclusions made from the results of this thesis. The viability of the chosen methods to speed up metadata extraction will be decided, and the methods used to accomplish the goals of the thesis will be discussed and evaluated.

Several areas containing potential future work, both in this particular project, and the field as a whole, have been identified. The identified future work is presented, and interested readers can use this section to identify natural continuations of this thesis.

7.1 Conclusion

The hypothesis that multiple stages of mining yields a shorter time required to mine metadata was correct. By first mining easily accessible metadata, and then advancing on to more detailed metadata the system appears more responsive than previously.

The *short-circuiting* feature introduced in this thesis ensures fast access to arbitrary first results, which are desirable when, for example a USB device has just been inserted in to an IVI system.

The Tracker mining software appears to be a good base to build an IVI multimedia mining software upon. Tracker was relatively easy to modify, and is designed with extensibility in mind.

7.2 Discussion of working method and project

We believe choosing an existing software to modify, rather than creating an entirely new software, was a good choice. The time required to understand the Tracker software was underestimated, but we believe the end result is much better than it would have been if the entire software was developed from scratch.

Much time was spent on comparing the Nepomuk-KDE project and the Tracker project. Nepomuk-KDE appears to use much more attractive technologies due to in-house expertise at Pelagicore, such as Qt and C++, rather than Tracker's C and GLib. By benchmarking and testing both Nepomuk-KDE and Tracker, we saw that Nepomuk-KDE was unusable for our purposes. We believe the time spent on benchmarking saved us a lot of time in the long run.

Tracker has been under development since 2005, has a complex structure, and is divided into many modules, which took a long time to understand. In hindsight, we believe we made the correct choice in basing our software on Tracker rather than trying to replicate many of its features, which would not have fitted in the time frame for this thesis.

Pelagicore as a company is profiled as an open source company, and does a lot of work with open source software and components, therefore we also think it is more in the spirit of the company to use and extend an existing software, rather than developing an entirely new solution.

7.3 Recommendations for future work

This project converted SPARQL expressions to SQL expressions and used the resulting expressions with a relational database. Other kinds of databases, such as triple stores or graph stores may be more efficient for storing this kind of data, and it would be interesting to see any difference in performance when comparing the relational database used in this project with other kinds of databases.

A global state indicating properties such as the current indexing stage would make it easier to decide on stage-specific actions in Metadex. It would be interesting to see any possible performance gains when the entire system is aware of the current mode of operation, such as when the system is short-circuited, running in the first stage of mining, or doing a complete metadata extraction.

It should be investigated why Metadex is slower than the original Tracker implementation when re-indexing previously seen media, and new methods for quickly recognizing previously indexed media should be researched. The current implementation relies fully on comparing the modification times of the files in the directory being indexed with an old stored modification time. The stored modification time could be stored in a way such that it is more quickly accessible, or a different method could be used all together, such as a very fast hash of the entire device.

The image extractors are much more unreliable than the extractors for audio or video in terms of predictability and stability. The image extractors use an unpredictable amount of time when compared to the other extractors, for the same number of files. It should be possible to achieve similar stability when extracting metadata from images as when extracting metadata from audio and video.

More stages of mining should be added, such as stages which connect to the internet in order to find more metadata. Online databases such as MusicBrainz[1], Gracenote[2]

[1]see: `http://www.musicbrainz.org`

[2]see: `http://www.gracenote.com/`

and IMDB[3] can be used to provide information such as the cast of a movie, or the lyrics of a song.

[3]see: `http://www.imdb.com`

Bibliography

[1] M. A. R. Megan Bayly, Kristie L. Young, Sources of distraction inside the vehicle and their effects on driving performance, in: Driver Distraction: Theory, Effects, and Mitigation, Taylor & Francis, 2008, pp. 192–210.

[2] M. A. Perez, Safety implications of infotainment system use in naturalistic driving, Work 41 (Supplement 1 / 2012) (2012) 5815–5818.

[3] W. A. Bhat, S. M. K. Quadri, A quick review of on-disk layout of some popular disk file systems, Global Journal of Computer Science and Technology XI (VI Version I).

[4] Microsoft Corporation, Microsoft Extensible Firmware Initiative FAT32 File System Specification, version 1.03 Edition (December 2000).

[5] B. A. Myers, The importance of percent-done progress indicators for computer-human interfaces, in: CHI '85: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM, New York, NY, USA, 1985, pp. 11–17.
URL `http://dx.doi.org/10.1145/317456.317459`

[6] A. Bernardi, G. Grimnes, T. Groza, S. Scerri, The NEPOMUK Semantic Desktop, in: P. Warren, J. Davies, E. Simperl (Eds.), Context and Semantics for Knowledge Management, Springer Berlin Heidelberg, 2011, pp. 255–273.
URL `http://dx.doi.org/10.1007/978-3-642-19510-5_13`

[7] M. Uschold, M. Gruninger, M. Uschold, M. Gruninger, Ontologies: Principles, methods and applications, Knowledge Engineering Review 11 (1996) 93–136.

[8] T. R. Gruber, A translation approach to portable ontology specifications, Knowl. Acquis. 5 (2) (1993) 199–220.
URL `http://dx.doi.org/10.1006/knac.1993.1008`

[9] G. Smethurst, Changing the In-Vehicle infotainment landscape, Tech. rep., GENIVI Alliance (2010).

[10] B. Perens, et al., The open source definition, Open sources: voices from the open source revolution (1999) 171–85.

[11] J. Lovejoy, Understanding the three most common open source licenses, Tech. rep., OpenLogic (2012).

[12] P. Vescuso, A. Dalrymple, Gplv3 licenses quadruple in 2009, but gpl projects drop by five percent from 2008 levels, Tech. rep., Black Duck Software (2009).

[13] Gnu general public license, version 2, `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html` (June 1991).

[14] Gnu lesser general public license, version 2.1, `http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html` (February 1999).

[15] Merriam-Webster.com, ”meta” (2013).
URL `http://www.merriam-webster.com/dictionary/meta-?show=1&t=1366198122`

[16] NISO Press, Understanding Metadata, National Information Standards Organization Press, 2004.
URL `http://www.niso.org/publications/press/UnderstandingMetadata.pdf`

[17] Dublin Core, Dublin Core Metadata Element Set, Version 1.1: Reference Description, Dublin Core Metadata Initiative, 2012.
URL `http://dublincore.org/documents/2012/06/14/dces/`

[18] E. Fleischman, Advanced streaming format (asf) specification, Tech. rep., Microsoft Corporation (January 1998).
URL `http://tools.ietf.org/html/draft-fleischman-asf-00`

[19] Advanced systems format (asf) specification, Tech. rep., Microsoft Corporation (January 2012).

[20] The ogg encapsulation format, Tech. rep., The Internet Society (May 2003).
URL `http://www.ietf.org/rfc/rfc3533.txt`

[21] Vorbis i specification, Tech. rep., Xiph.org Foundation (February 2012).
URL `http://www.xiph.org/vorbis/doc/Vorbis_I_spec.html`

[22] Theora specification, Tech. rep., Xiph.org Foundation (March 2011).
URL `http://www.theora.org/doc/Theora.pdf`

[23] Information technology — coding of audio-visual objects — part 3: Audio, Tech. rep. (December 2005).

[24] Information technology — coding of audio-visual objects — part 12: Iso base media file format, Tech. rep. (Oktober 2008).

[25] Information technology — coding of audio-visual objects — part 14: Mp4 file format, Tech. rep. (November 2003).

[26] JEITA, Digital Still Camera Image File Format Standard (Exchangeable image file format for Digital Still Cameras: Exif) Version 2.1, JEITA, 1998.

[27] I. T. Union, Information technology - digital compression and coding of continuous-tone still images - requirements and guidelines, Tech. rep., CCITT (1993).
URL `http://www.w3.org/Graphics/JPEG/itu-t81.pdf`

[28] Adobe Systems Incorporated, TIFF Specification, Tech. rep. (1992).
URL `http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf`

[29] Michael Steidl, Photo Metadata White Paper 2007, Tech. rep. (2007).
URL `http://www.iptc.org/std/photometadata/0.0/documentation/IPTC-PhotoMetadataWhitePaper2007_11.pdf`

[30] Adobe Systems Incorporated, XMP Adding Intelligence to Media, Tech. rep. (2005).
URL `http://partners.adobe.com/public/developer/en/xmp/sdk/XMPspecification.pdf`

[31] M. Nilsson, Id3v2.3.0 informal standard, Tech. rep. (February 1999).
URL `http://id3.org/d3v2.3.0`

[32] F. Manola, E. Miller (Eds.), RDF Primer, W3C Recommendation, World Wide Web Consortium, 2004.
URL `http://www.w3.org/TR/rdf-primer/`

[33] S. Harris, A. Seaborne (Eds.), SPARQL 1.1 Query Language, W3C Recommendation, World Wide Web Consortium, 2013.
URL `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/`

[34] GNOME Team, GLib Reference Manual, 2013.
URL `https://developer.gnome.org/glib/2.36/`

[35] H. Pennington, A. Carlsson, A. Larsson, S. Herzberg, S. McVittie, D. Zeuthen (Eds.), D-Bus Specification, freedesktop.org, 2013.
URL `http://dbus.freedesktop.org/doc/dbus-specification.html`

[36] Beagle-team, Beagle, page is now offline, only available through archiving services (2008).
URL `http://web.archive.org/web/20080708182518/http://www.beagle-project.org/Development`

A

Full size graphs from results chapter

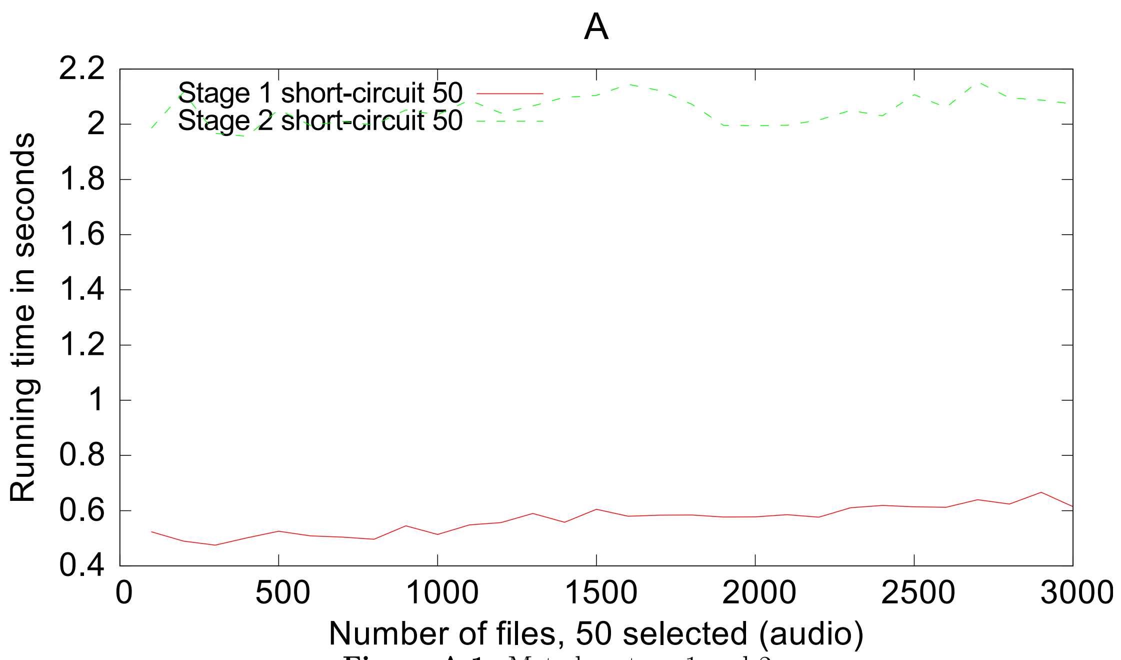


Figure A.1: Metadex stage 1 and 2

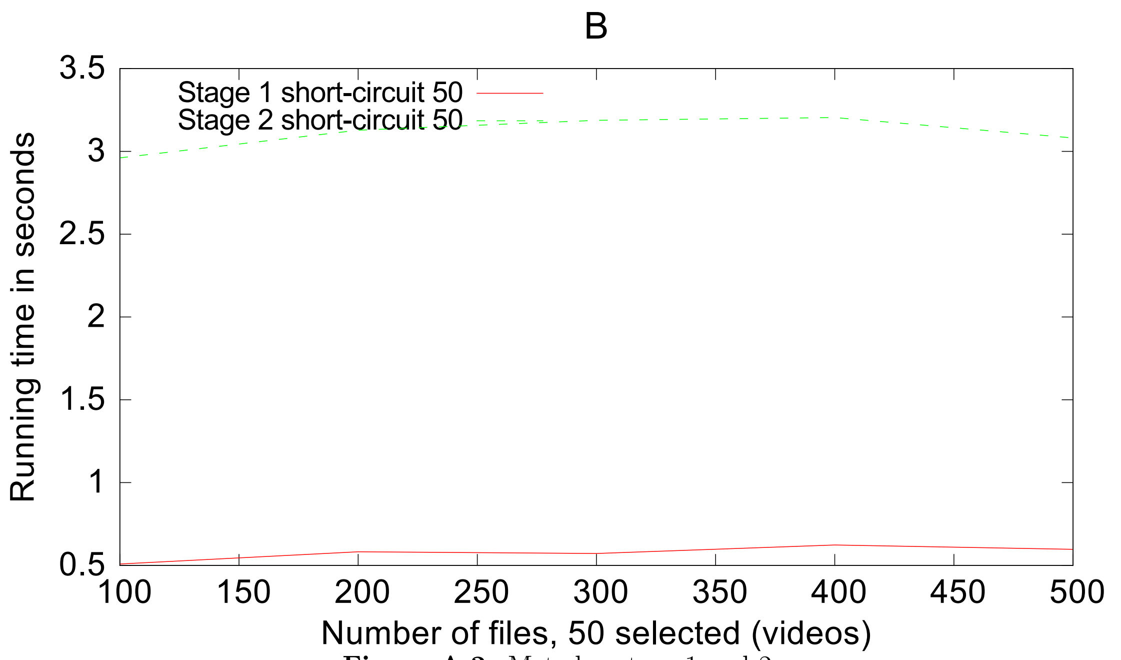


Figure A.2: Metadex stage 1 and 2

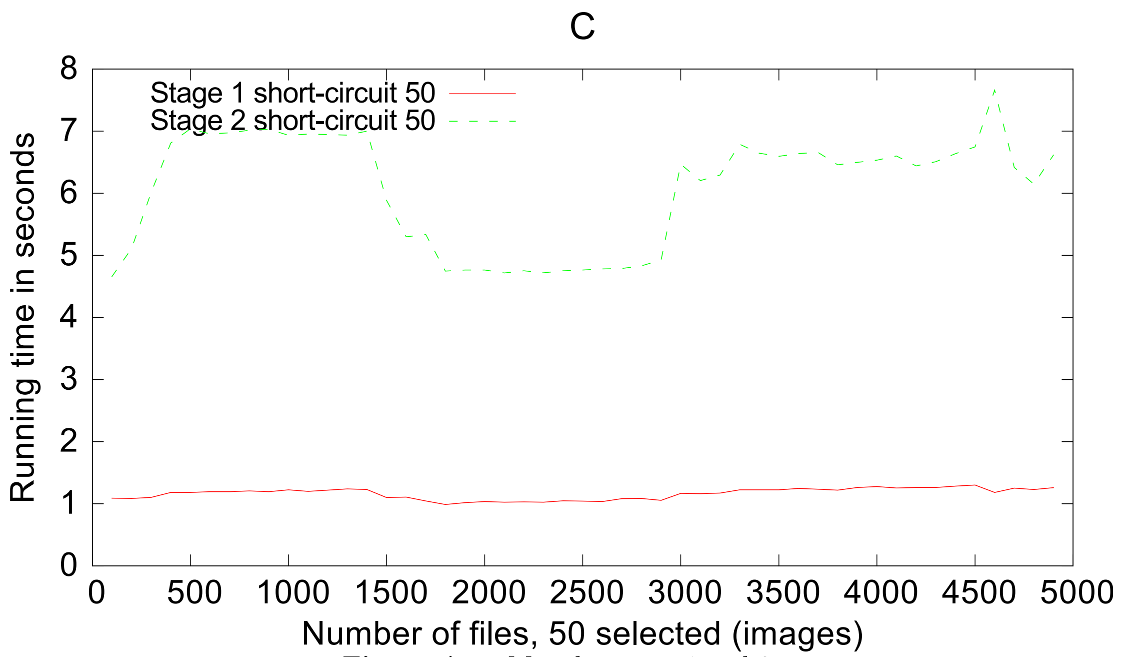


Figure A.3: Metadex stage 1 and 2

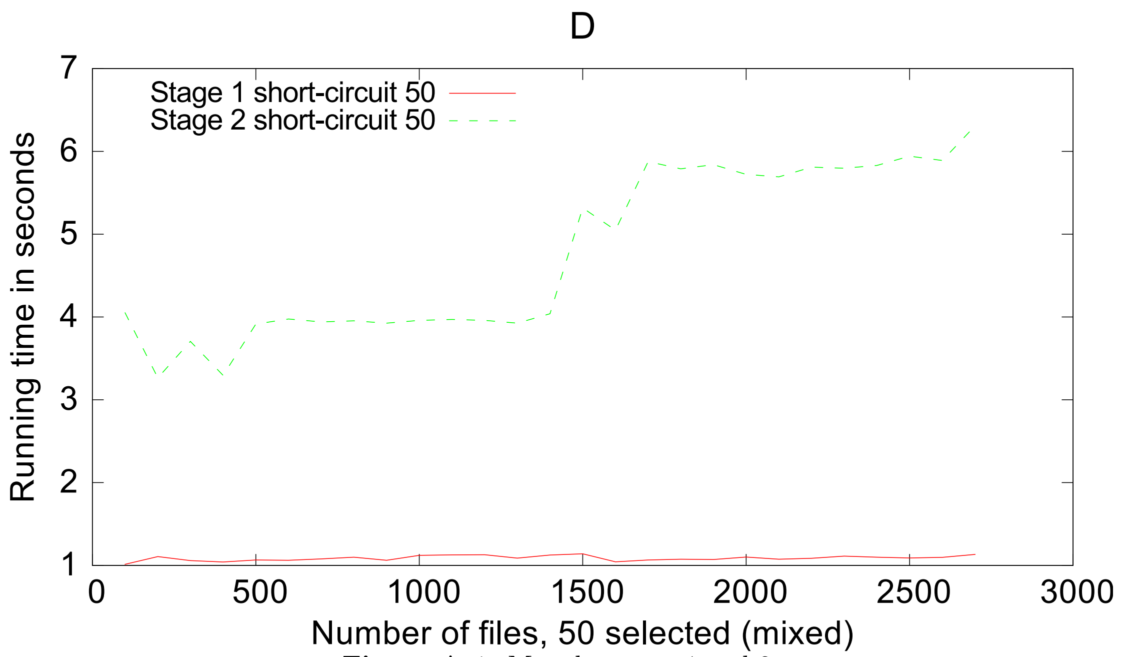


Figure A.4: Metadex stage 1 and 2

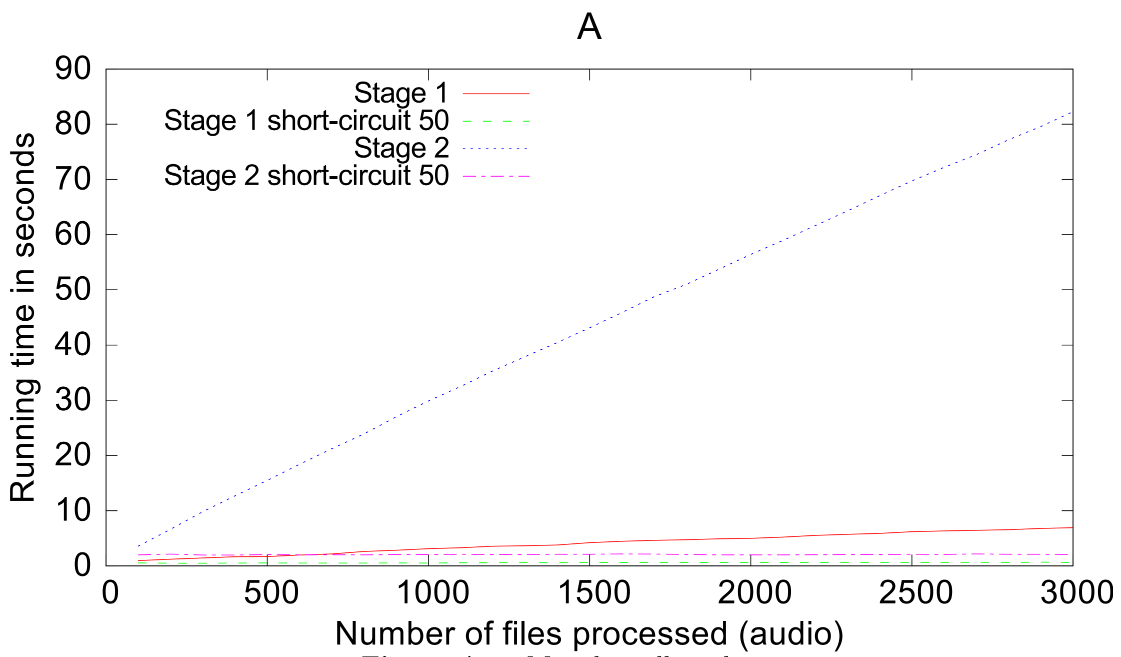


Figure A.5: Metadex, all modes

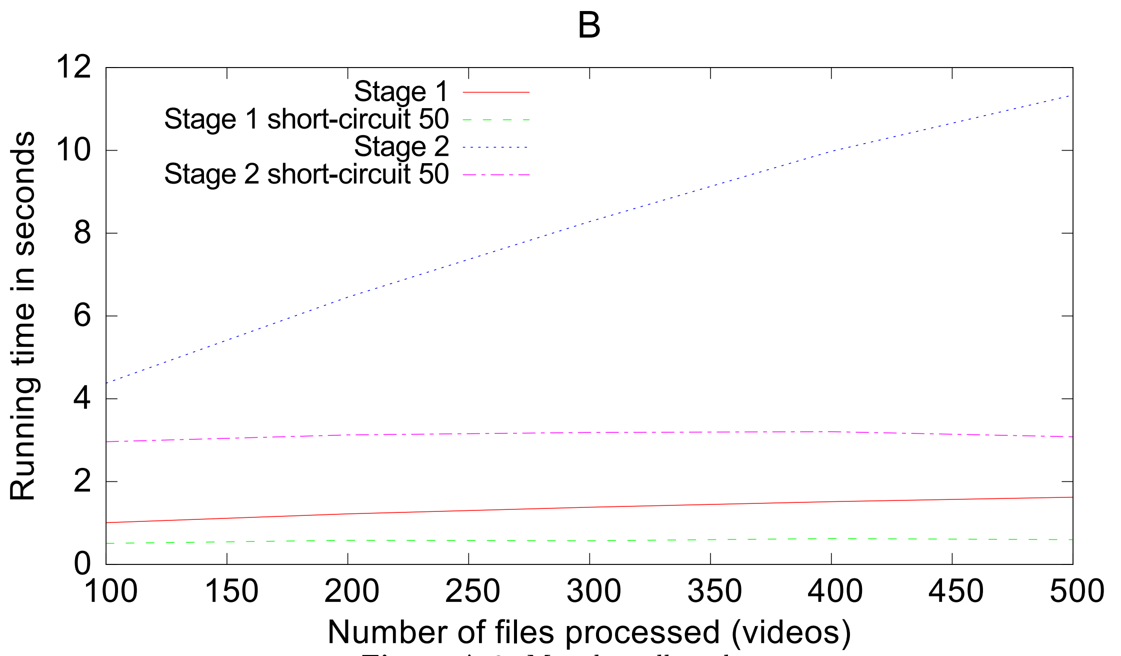


Figure A.6: Metadex, all modes

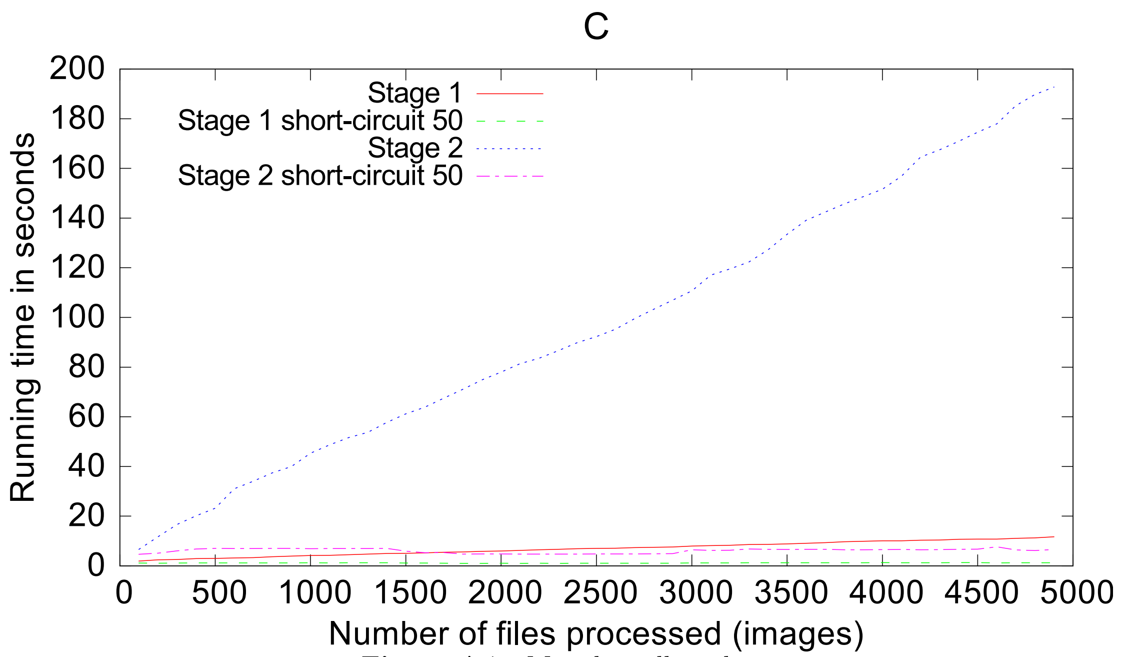


Figure A.7: Metadex, all modes

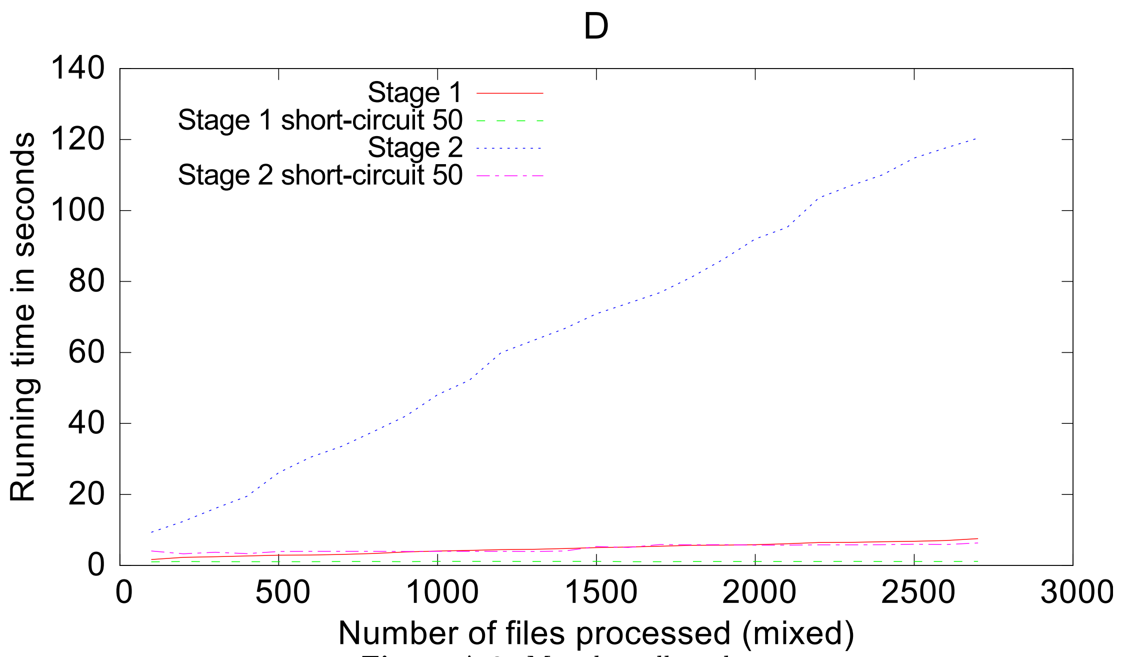


Figure A.8: Metadex, all modes

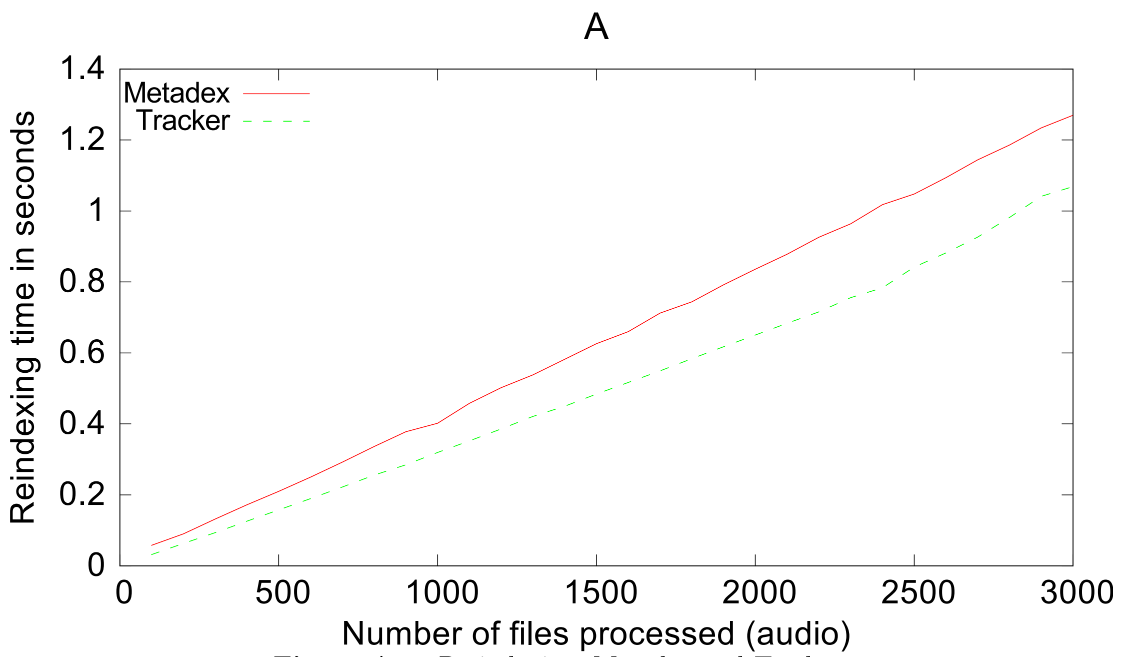


Figure A.9: Re-indexing, Metadex and Tracker

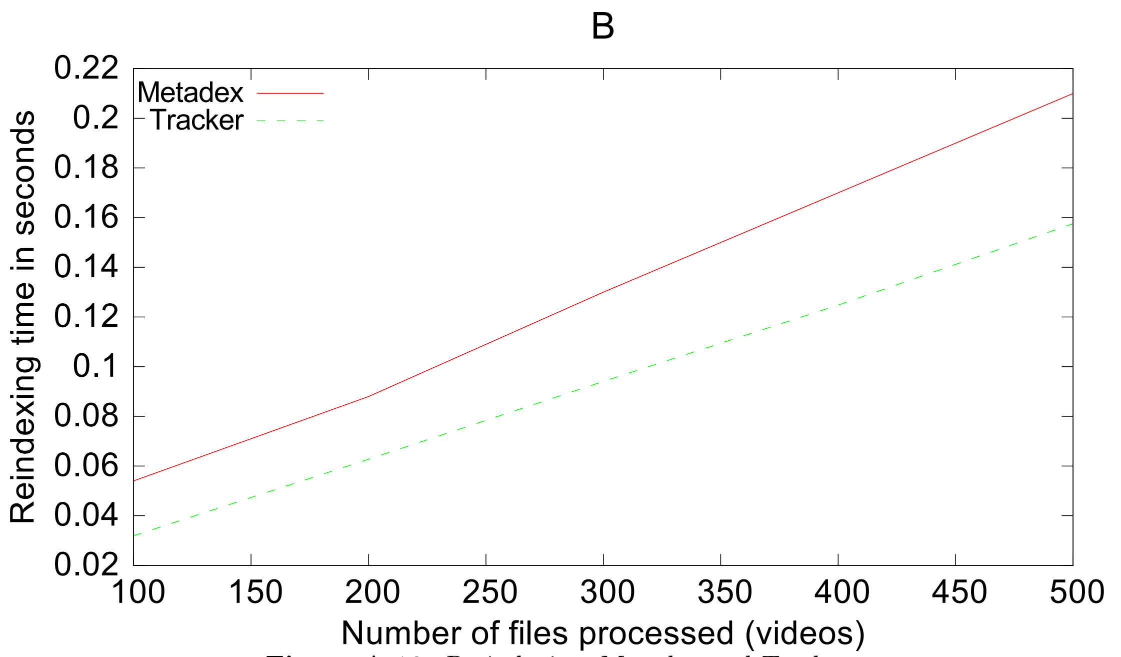


Figure A.10: Re-indexing, Metadex and Tracker

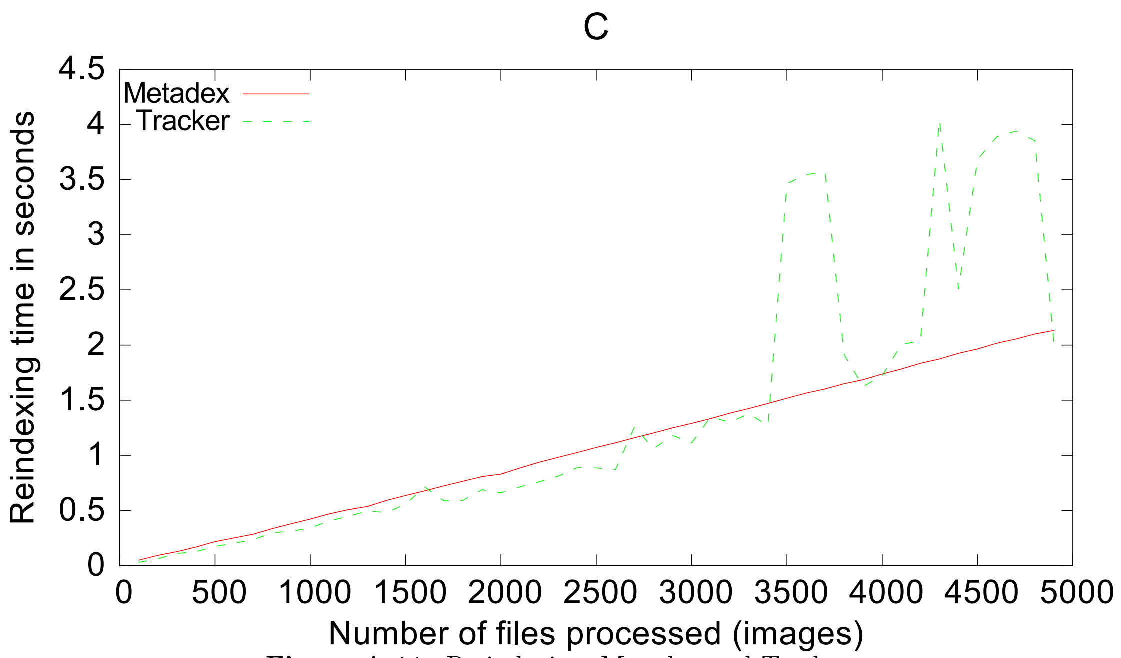


Figure A.11: Re-indexing, Metadex and Tracker

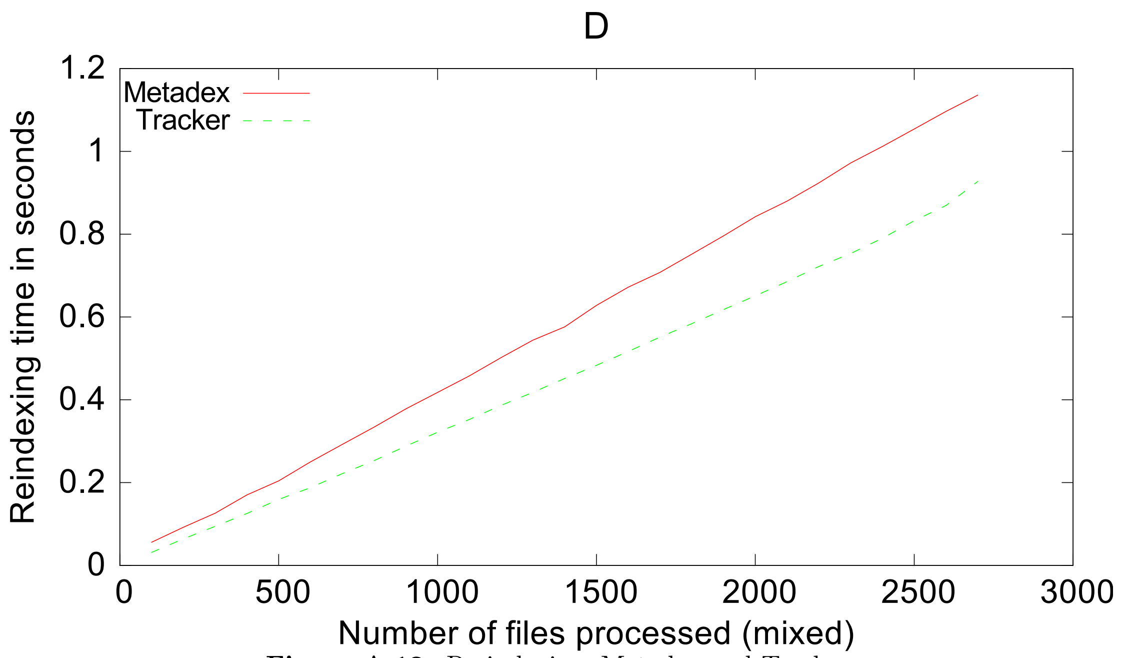


Figure A.12: Re-indexing, Metadex and Tracker

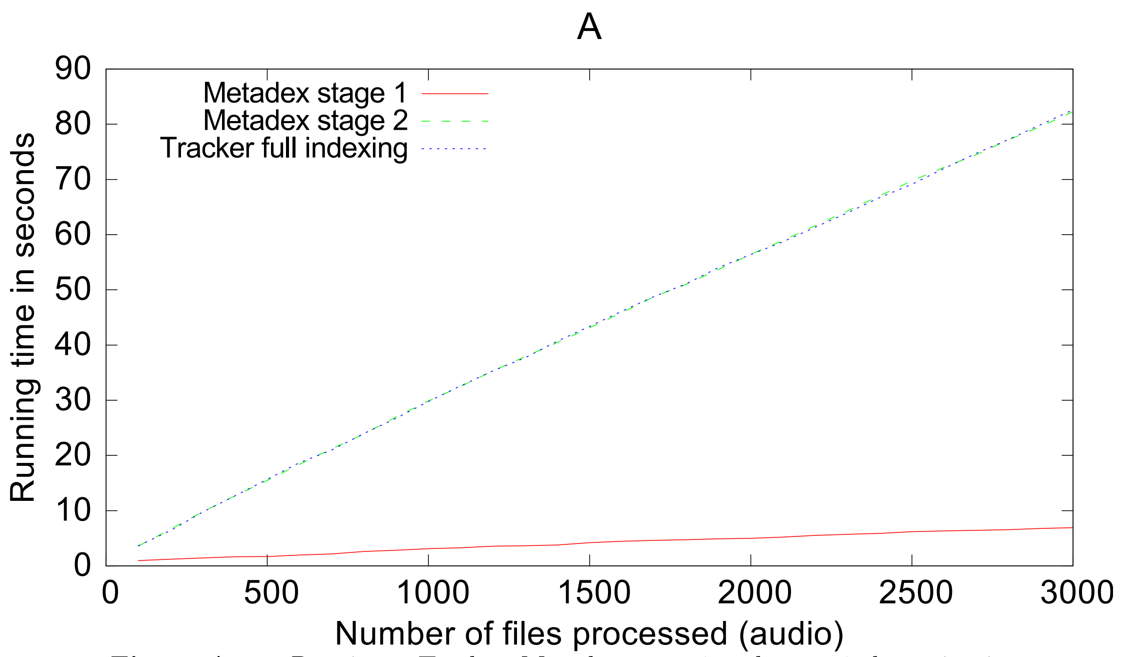


Figure A.13: Runtimes, Tracker, Metadex stage 1 and stage 1 short-circuit

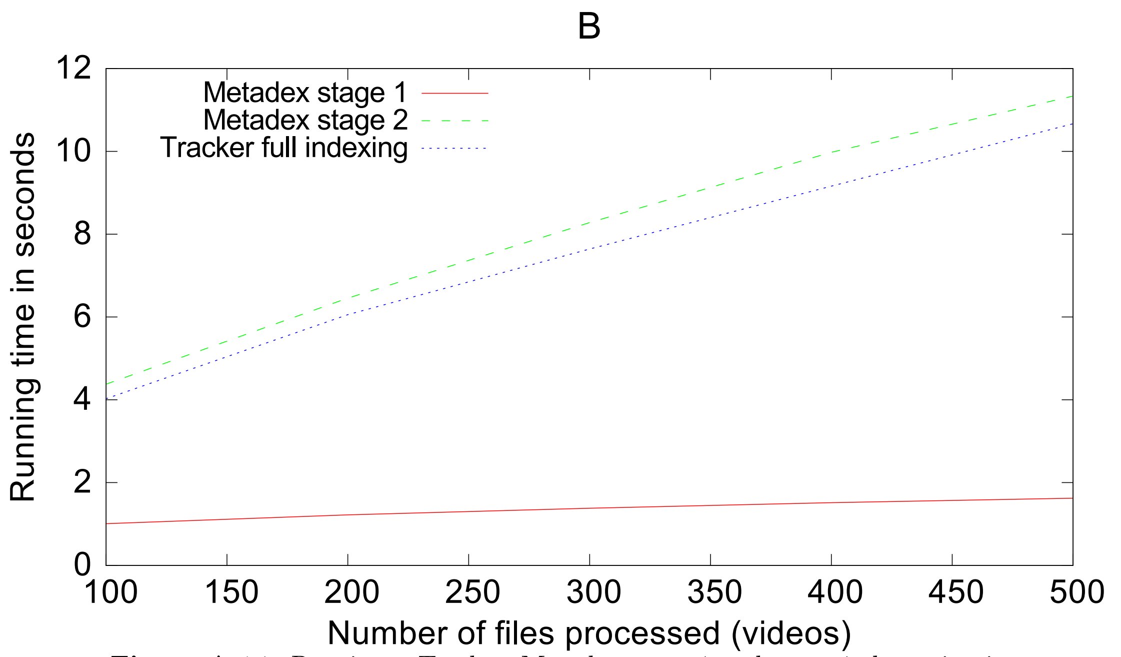


Figure A.14: Runtimes, Tracker, Metadex stage 1 and stage 1 short-circuit

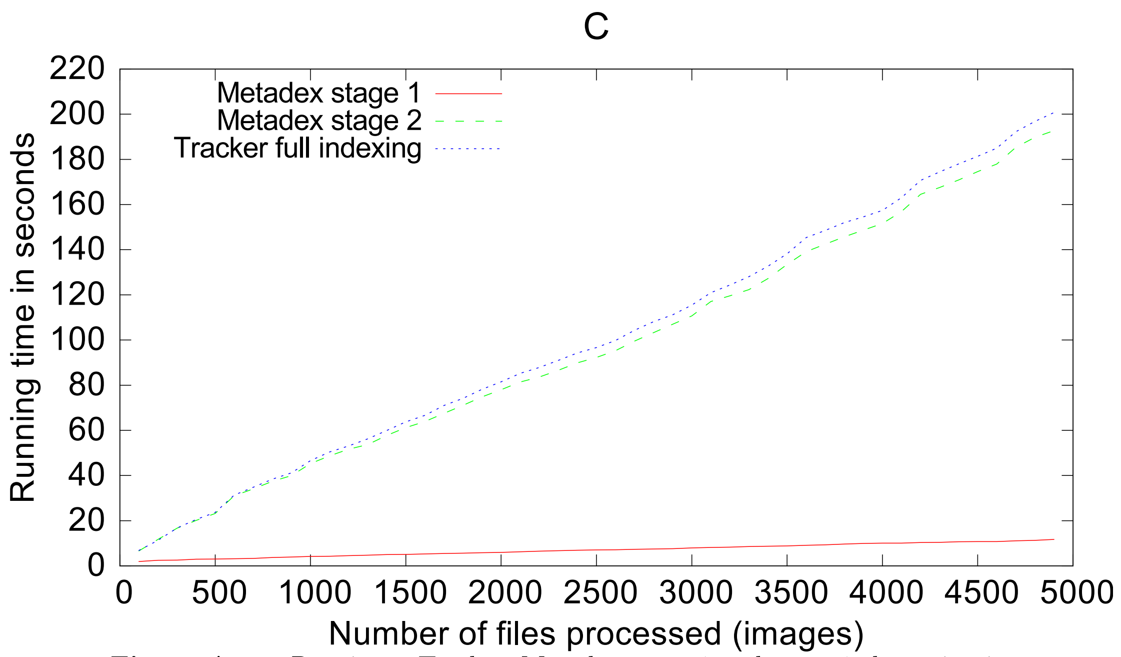


Figure A.15: Runtimes, Tracker, Metadex stage 1 and stage 1 short-circuit

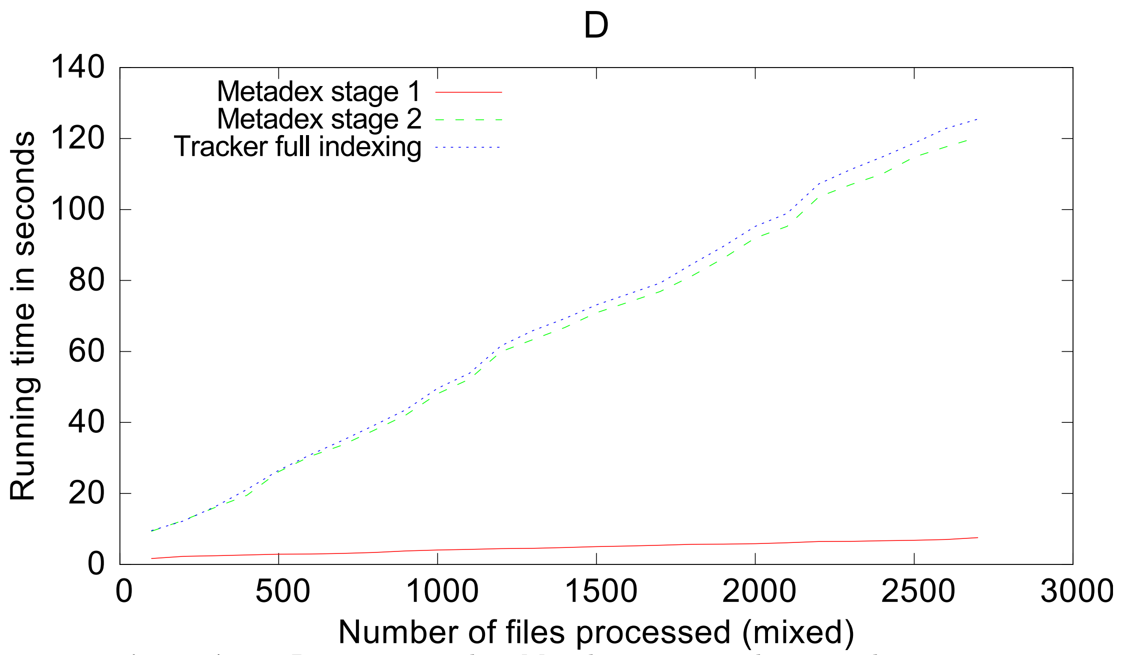


Figure A.16: Runtimes, Tracker, Metadex stage 1 and stage 1 short-circuit

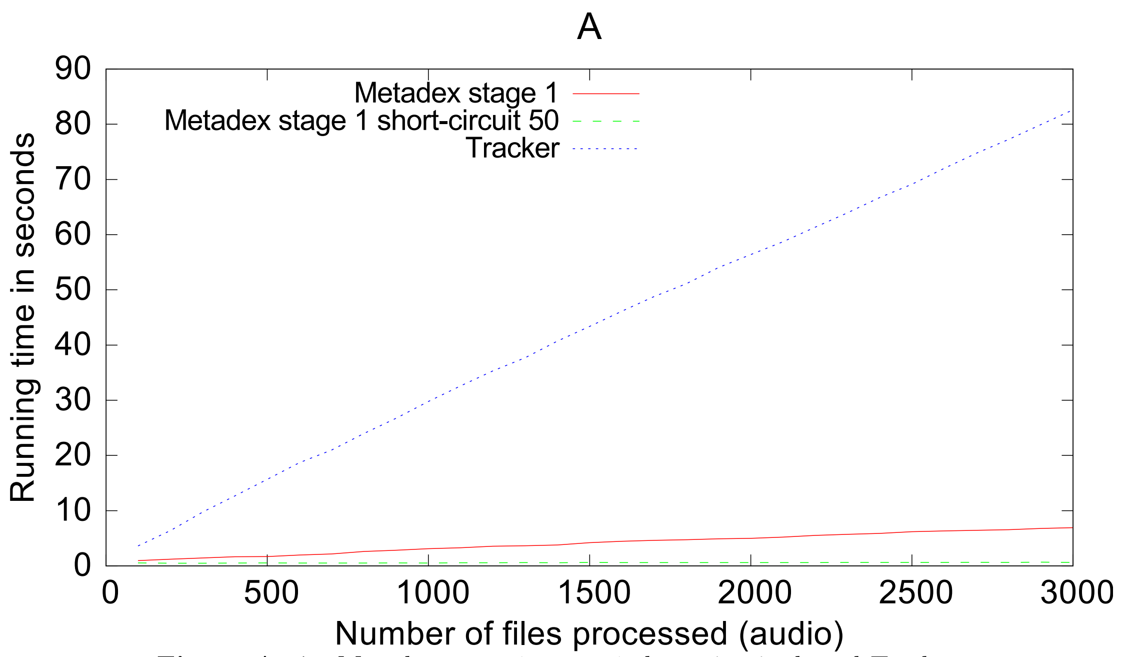


Figure A.17: Metadex stage 1, stage 1 short-circuited, and Tracker

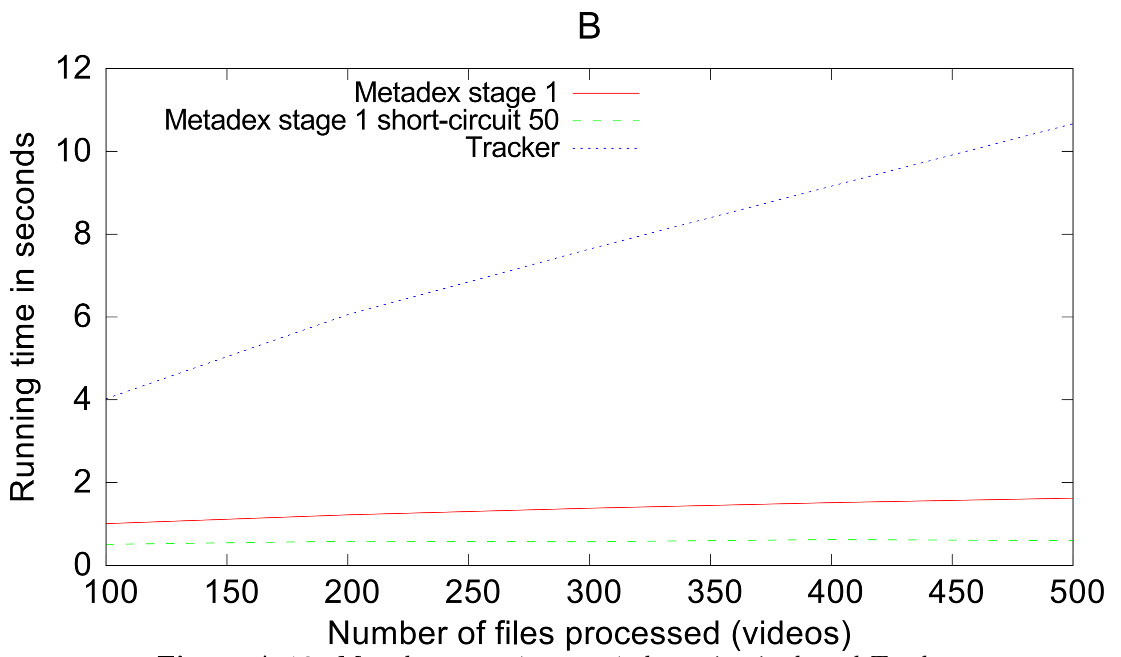


Figure A.18: Metadex stage 1, stage 1 short-circuited, and Tracker

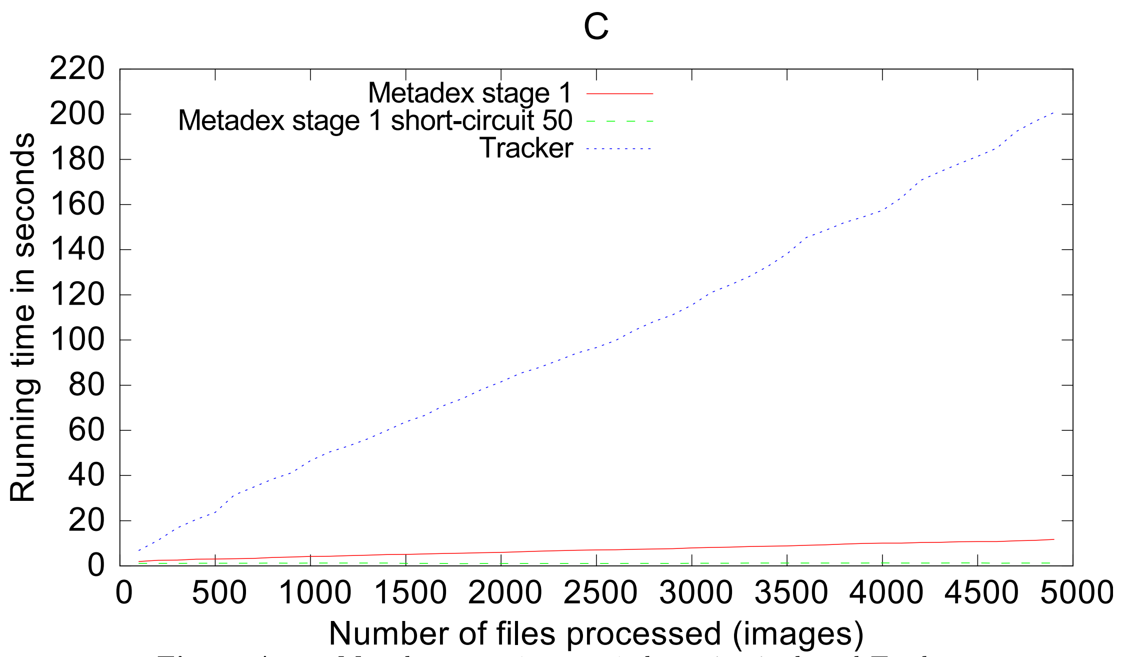


Figure A.19: Metadex stage 1, stage 1 short-circuited, and Tracker

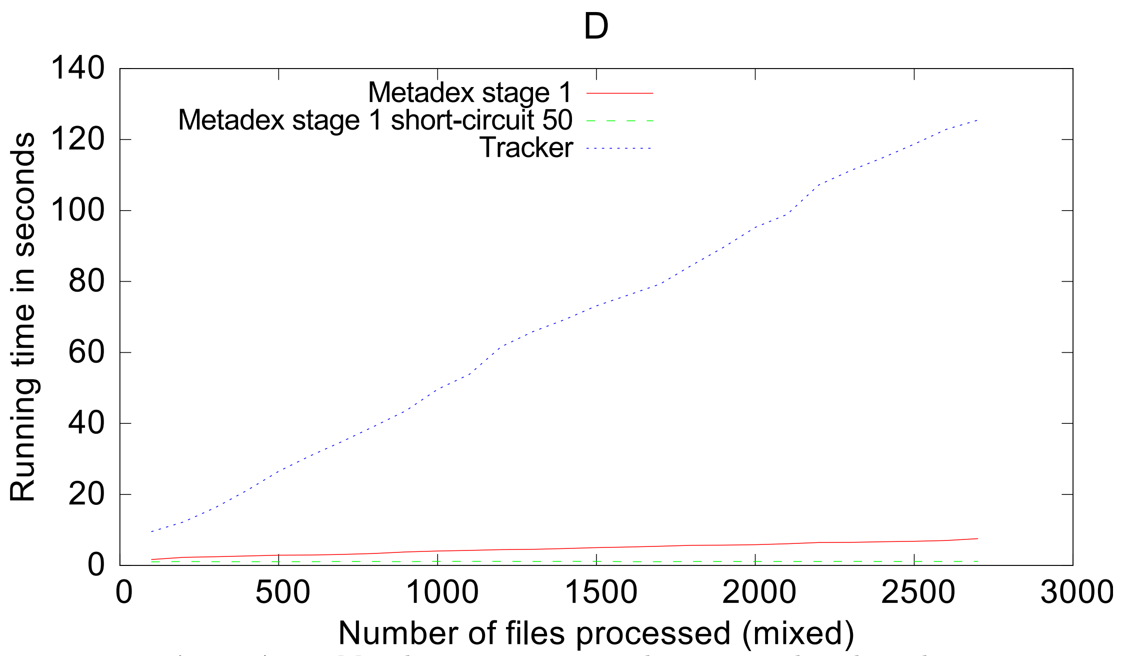


Figure A.20: Metadex stage 1, stage 1 short-circuited, and Tracker

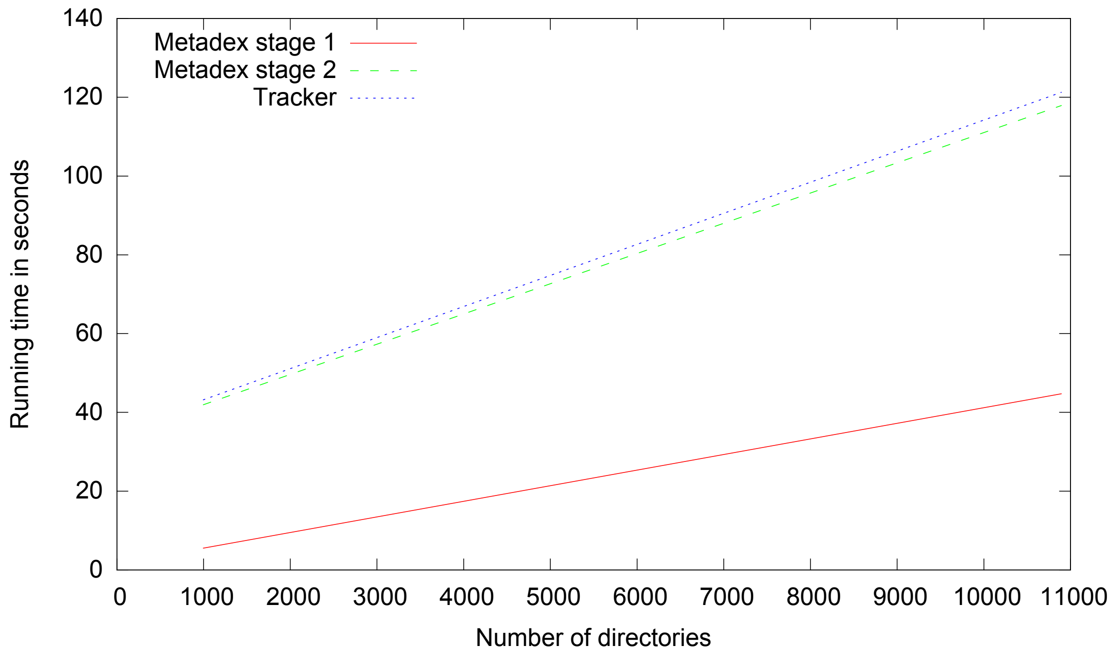


Figure A.21: Runtimes, varying directory structure depth

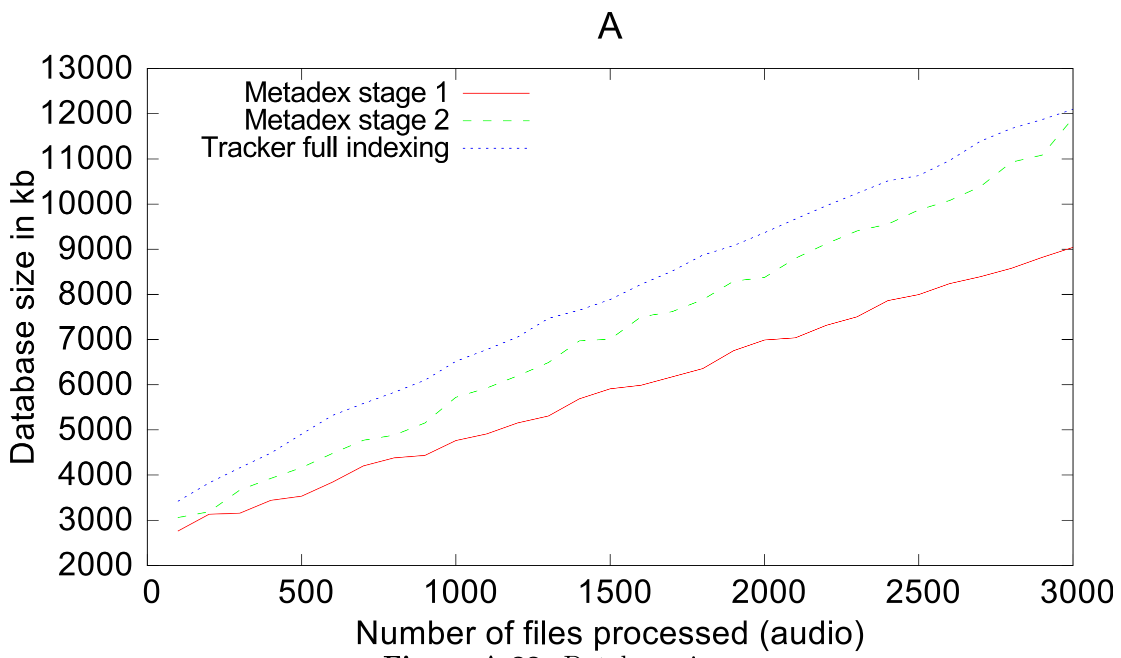


Figure A.22: Database sizes

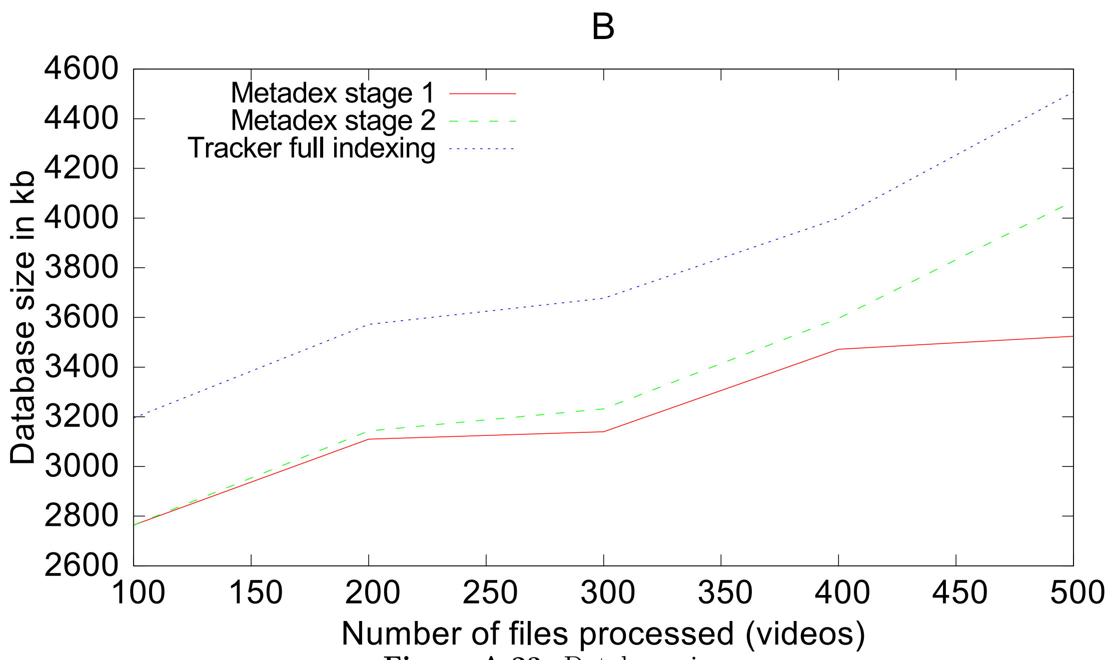


Figure A.23: Database sizes

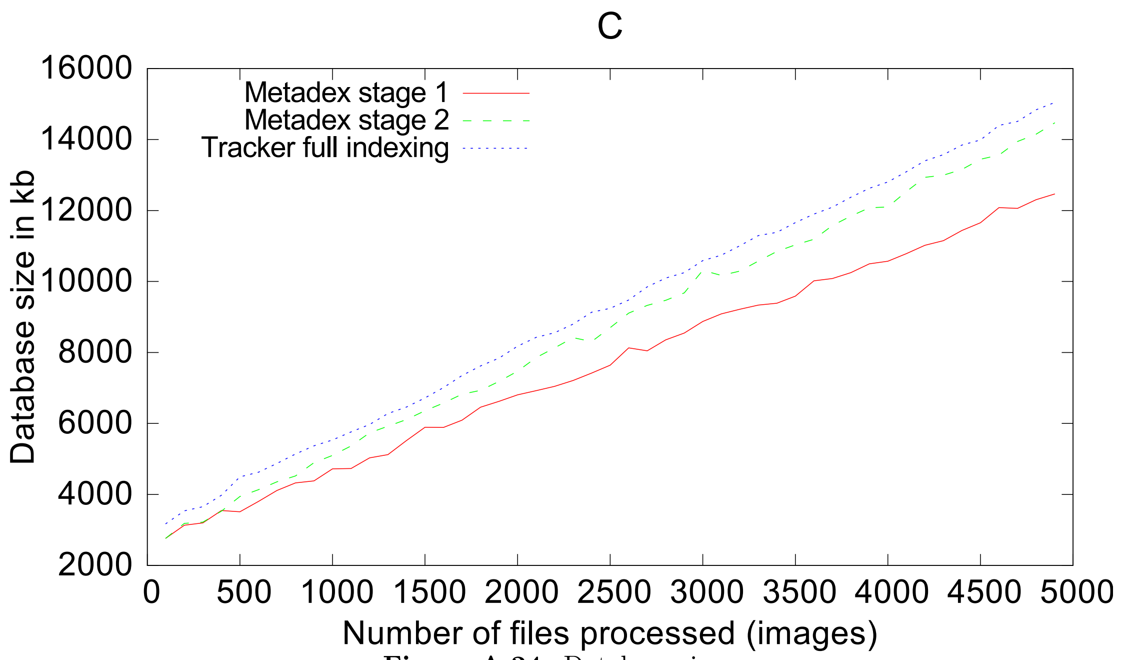


Figure A.24: Database sizes

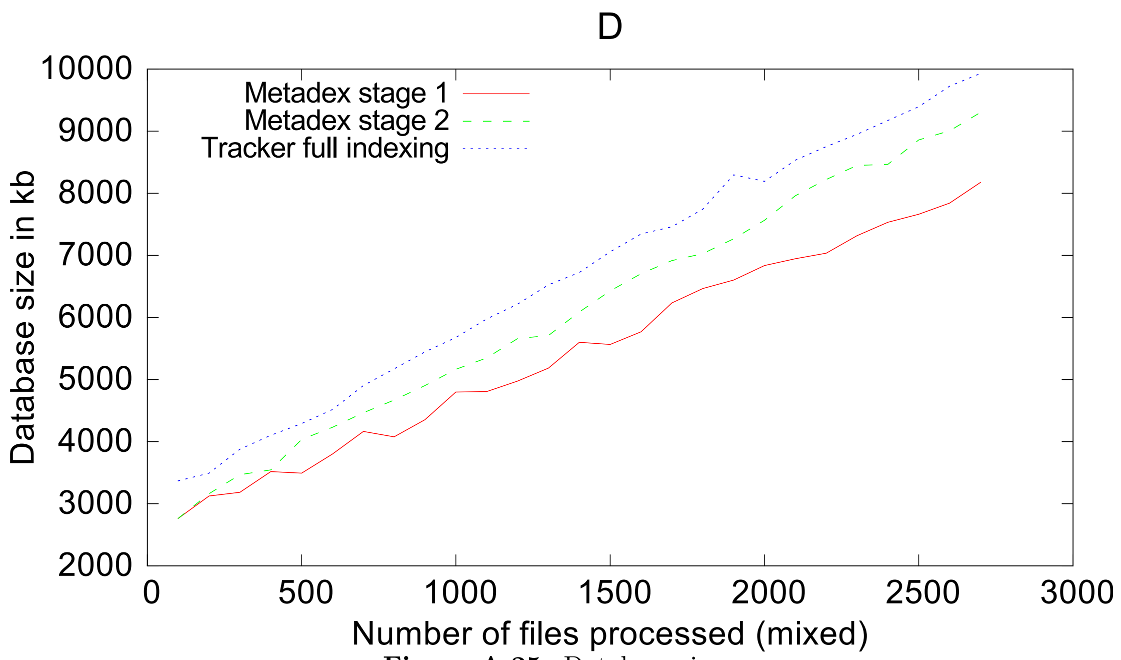


Figure A.25: Database sizes

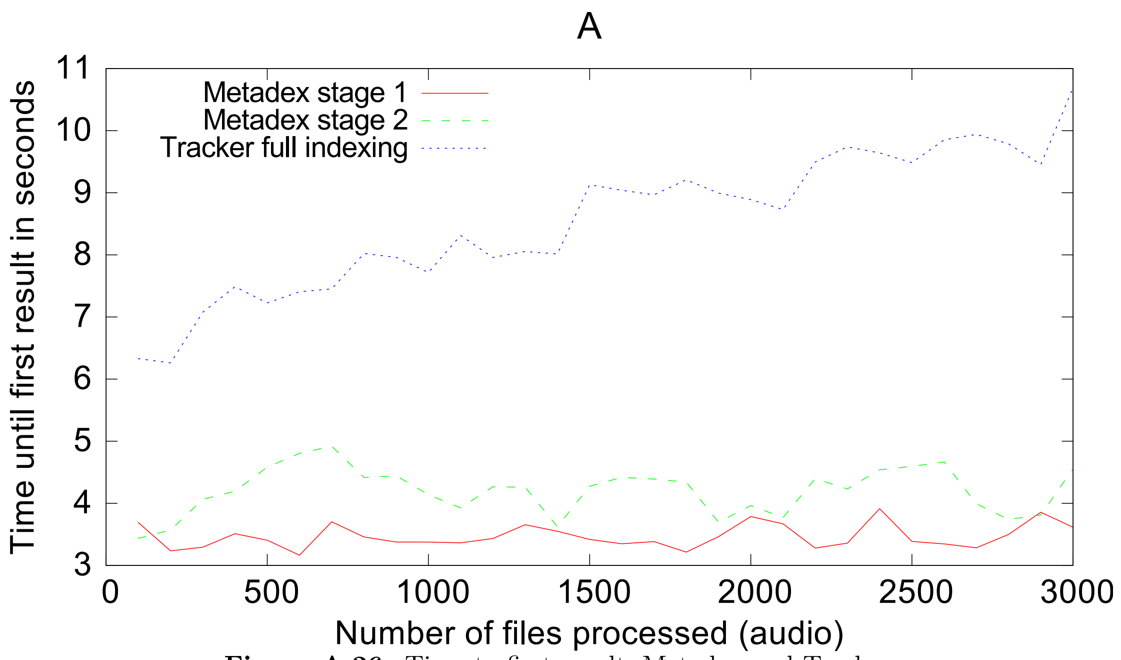


Figure A.26: Time to first result, Metadex and Tracker

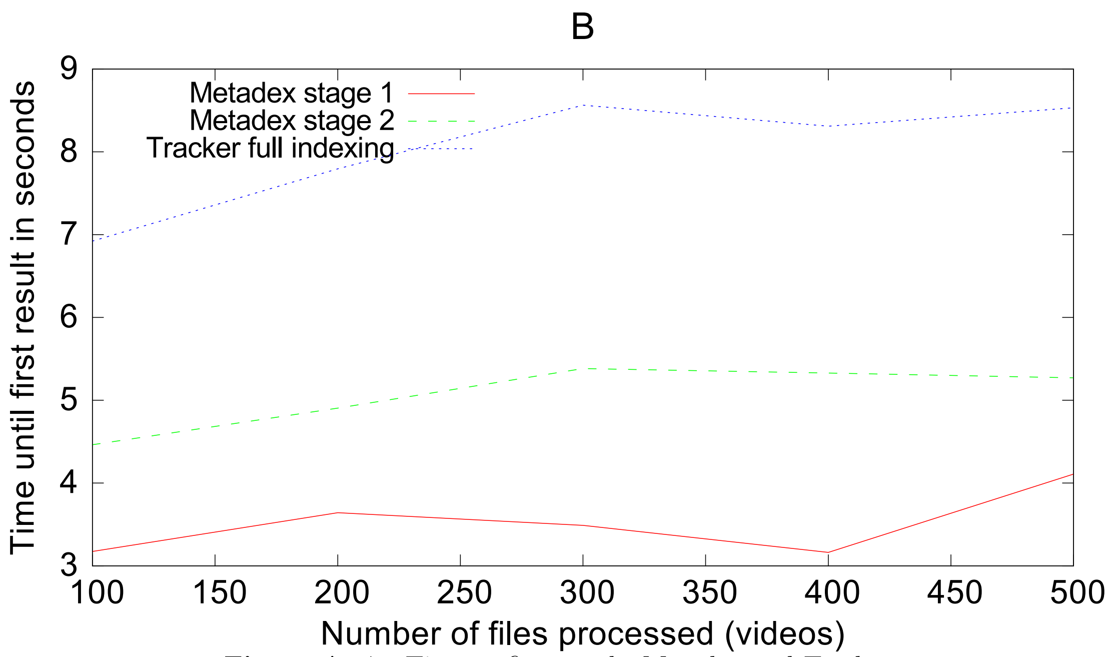


Figure A.27: Time to first result, Metadex and Tracker

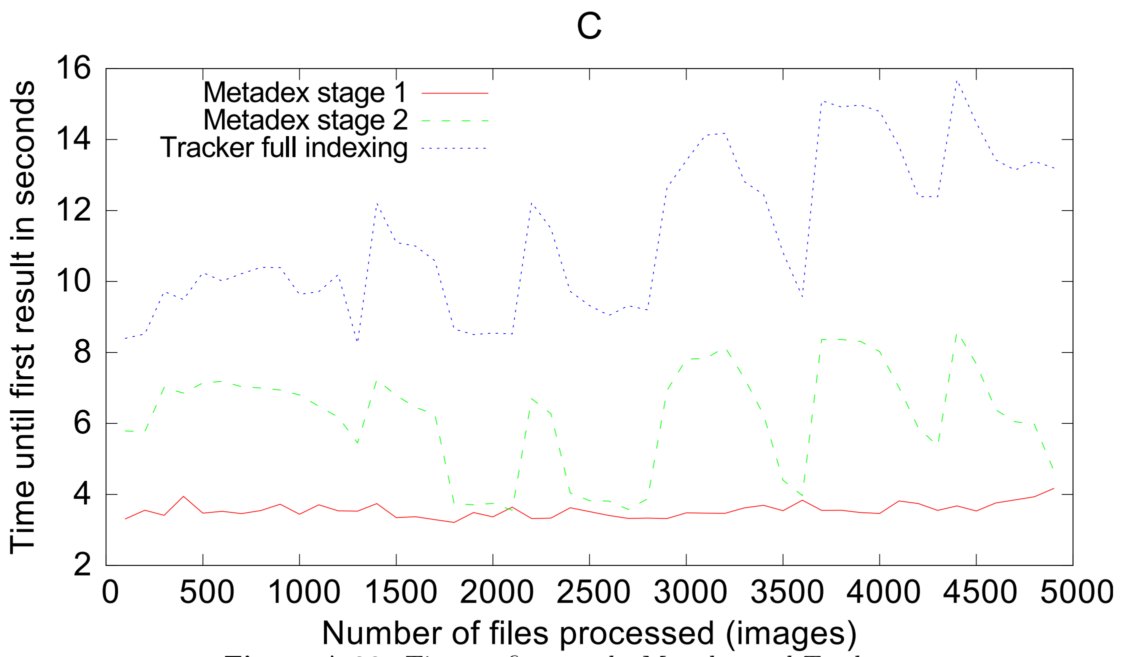


Figure A.28: Time to first result, Metadex and Tracker

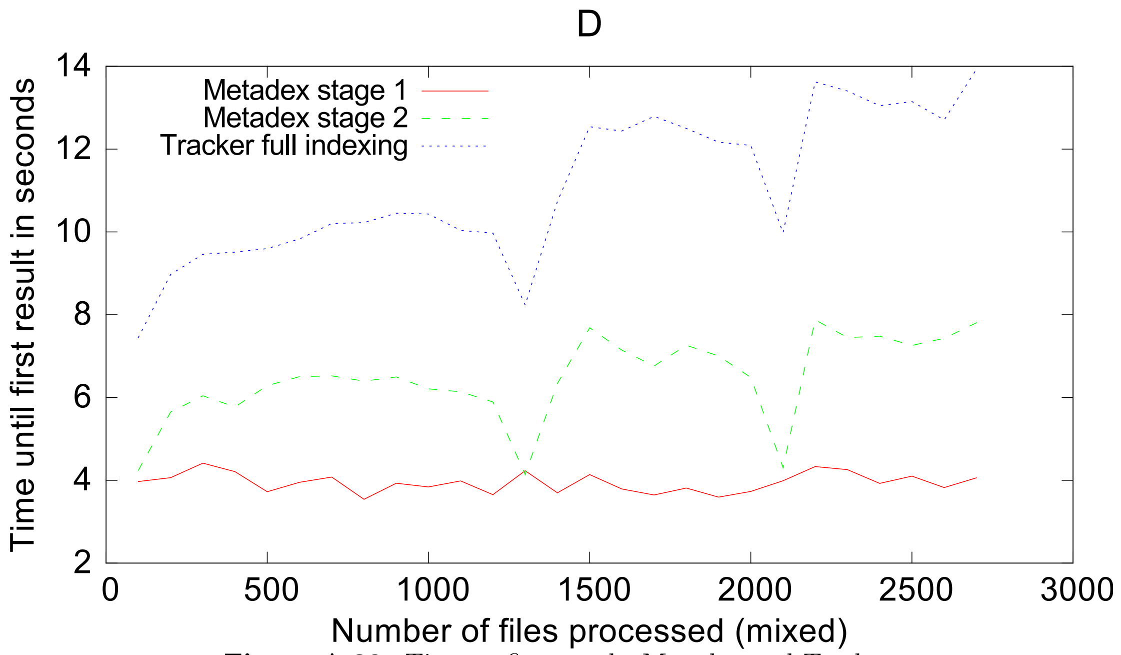


Figure A.29: Time to first result, Metadex and Tracker