EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Resource constrained meta-data storage and retrieval

van den Broek, K.H.M.

Award date:
2005

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Resource constrained meta-data storage and retrieval

ing. K.H.M. van den Broek

Supervisors:

dr. A.T.M. Aerts (TU/e)
ir. J.P. van Gassel (Philips)
ir. A. Sinitsyn (Philips)
dr. S.R. Cumpson (Philips)

Eindhoven, October 2005

Abstract

In the Personal Infotainment Companion project at Philips Research, a new class of portable audio, video and photo devices is being developed for the consumer market. When vast amounts of multimedia files are accumulated on these devices, an efficient search mechanism is required. A Database Management System is an important part in such a mechanism. Due to the resource limitations of embedded devices, choosing the right DBMS is an important aspect of the PIC design. In order to select a good DBMS, requirements are defined along with several use-cases. Database concepts are investigated and a benchmark framework is designed and implemented. Finally, several DBMSs are benchmarked and analyzed. The results show both the strengths and limitations of current-generation DBMSs.

Acknowledgments

I carried out my graduation project within the Storage Systems & Applications group at Philips Research Eindhoven. I received guidance from my supervisors Joep van Gassel and Stephen Cumpson. I'd also like to thank Alexander Sinitsyn for technical guidance.

I very much appreciated the help of Ard, Eddy and many other colleagues on C++ programming, in which I was an absolute beginner. Not to forget, also the coffee vending machine made an invaluable contribution to my progress.

From the start until the finish line, I was supervised by Ad Aerts from the TU/e, who took the time to review, criticize and provide new insights in the endless stream of thesis concepts that I sent.

Sometimes last, but never least, I'd like to thank my girlfriend Suzan, my family and friends for bearing with me during my busy and sometimes grumpy days.

Contents

1 Introduction **6**
- 1.1 Problem statement . . . 6
- 1.2 Background . . . 7

2 Requirements **9**
- 2.1 Introduction . . . 9
- 2.2 Developer requirements . . . 9
  - 2.2.1 MDB framework related . . . 9
  - 2.2.2 DBMS related . . . 10
- 2.3 User requirements . . . 10
- 2.4 Current Situation . . . 11
- 2.5 Two sub-projects . . . 12
- 2.6 Hardware target platforms . . . 13
  - 2.6.1 Target platform 1: mobile audio player . . . 13
  - 2.6.2 Target platform 2: mobile audio/video player . . . 13
  - 2.6.3 Target platform 3: home media server . . . 14
- 2.7 Scenarios . . . 14
- 2.8 Use-cases . . . 14

3 Basic concepts **16**
- 3.1 Introduction . . . 16
- 3.2 Meta-data . . . 16
  - 3.2.1 MPEG meta-data example . . . 17
- 3.3 Database concepts . . . 17
  - 3.3.1 Tree data structures . . . 18
  - 3.3.2 Berkeley DB . . . 20
  - 3.3.3 RDBMS . . . 23
  - 3.3.4 OODBMS . . . 26
  - 3.3.5 O/RMS . . . 27
  - 3.3.6 XML DBMS . . . 28
  - 3.3.7 DBMS summary . . . 29

3.4 Caching . . . 29
3.4.1 Definition of a cache . . . 30
3.4.2 Layers of cache . . . 30
3.4.3 Database benchmarking and cache . . . 30

4 Benchmarking and framework design 31
4.1 Introduction . . . 31
4.2 The benchmark definition . . . 31
4.3 Benchmarking properties . . . 32
4.4 Standards in benchmarking . . . 33
4.5 The benchmark procedure . . . 33
4.6 Defining the benchmarking area . . . 34
4.7 The actual benchmark . . . 35
4.8 Desired benchmark results . . . 35

5 Benchmarking framework implementation 37
5.1 Introduction . . . 37
5.2 Architecture . . . 38
5.3 Features . . . 40
5.4 Extensibility . . . 42
5.5 Debugging . . . 42
5.6 Benchmarks . . . 43

6 Benchmark result analysis 1 44
6.1 Introduction . . . 44
6.2 Benchmark parameters . . . 45
6.3 Testcase 1: database creation and filling . . . 46
6.4 Testcase 2: simple scan . . . 47
6.5 Testcase 3: simple scan with filter . . . 51
6.6 Testcase 4: simple scan with filter yielding no results . . . 51
6.7 Testcase 5: inner-join . . . 55
6.8 Testcase 6: inner-join with filter . . . 58
6.9 Testcase 7: simple scan with ordered results . . . 58
6.10 Aggregated results . . . 58

7 Benchmark result analysis 2 62
7.1 Benchmarking on the ARM architecture . . . 62
7.1.1 Observations and conclusions drawn from the results . . . 63
7.2 Database engines for MySQL . . . 67
7.3 Connection overhead . . . 69

8 Results and conclusions **71**
8.1 DBMS conclusions 71
8.1.1 DBMS recommendations per target platform 72

9 Future work and recommendations **73**
9.1 Introduction 73
9.2 Towards a more universal benchmarking framework 73
9.2.1 Client programming language independent benchmarking 73
9.3 Towards a more complete benchmarking framework 74
9.3.1 Supporting more operations 74
9.3.2 Measuring more dimensions 74
9.3.3 Measuring more environments 76
9.3.4 Identify cache influence 77
9.3.5 Parallel workloads 78
9.4 Towards a more accurate benchmarking framework 78
9.4.1 Dummy queries 78
9.5 Recommendations 78

Bibliography **79**

A Terms **80**

B Initial benchmarking framework experiments **81**
B.1 Introduction 81
B.2 Benchmark parameters 81
B.3 Testcase 1: database creation and filling 82
B.3.1 MDB file importer 85
B.3.2 Libdbi 86
B.4 Testcase 2: simple scan 87
B.5 Distortions 89
B.5.1 Other processes 89
B.5.2 Caching 90
B.5.3 Other Intermittent Distortions 90
B.6 Testcase 3: inner-join 90
B.7 Datasets 91

C MDB Analysis **94**
C.1 Introduction 94
C.1.1 Measurements 94
C.2 Conclusion 95

D MDB API and schemas **99**
D.1 Target platform 1, Mobile audio player 99
D.2 Target platform 2, Mobile audio/video player 99
D.3 Target platform 3, Home server 99

E Original project plan **103**
E.1 Introduction 103
E.2 Problem statement 104
E.3 Desired results 104
E.4 Time table 105
E.5 Risk evaluation 105

F Revised project plan **106**
F.1 Introduction 106
F.2 Problem statement 106
F.3 Desired results 107
F.4 New time table 107
F.5 Actual, historic time table 107
F.6 Risk evaluation 108

G Logbook **109**

Chapter 1

Introduction

This introduction presents a short overview of the document structure.

The remainder of this chapter briefly discusses the graduation assignment. The accompanying project plan including the time table is presented in appendix F.

In chapter 2, concrete, measurable and verifiable requirements are distilled from the graduation assignment.

Chapter 3 introduces several concepts that contribute to a better understanding of the problems and challenges in the database field.

In chapter 4, the different methods and industry standards to perform benchmarks are discussed.

Chapter 5 describes the implementation of an automated benchmarking framework. The benchmarking framework is deployed to measure the performance of several database management systems. Initial experiments to improve the correctness of the benchmarking framework are analyzed in appendix B. The first series of accurate measurements is described and analyzed in chapter 6. After an assessment of strengths and weaknesses of the benchmarked database management systems, a second series of benchmarks is performed in chapter 7. Here, the database management systems are individually tuned to highlight their strengths.

The benchmarking framework is a good base to build on. Ideas on expanding the framework for future benchmarks have been collected in chapter 9.

Chapter 8 concludes this graduation project and provides a view on what has been accomplished.

1.1 Problem statement

Philips is developing a new class of portable audio, video and photo devices for consumers within the PIC[1] [BvGS02] project. An important part of this project is the management of the meta-data about the multimedia. Therefore, Philips Research developed a framework for meta-data storage, modification and retrieval, called MDB[2] [WFS05].

A user application (e.g. the jukebox GUI) uses MDB to handle meta-data. For example: whenever the user wants to know what media created by "Madonna" is available on the device, MDB is invoked to find an answer to the question (the query). The results of this query are returned to the user application in the form of meta-data objects, called assets. These assets are currently stored using the MySQL RDBMS[3]. A DBMS provides operations to store, modify, search and retrieve data in a standardized and reliable way.

[1] Personal Infotainment Companion
[2] Multimedia DataBase
[3] Relational DataBase Management System

The earlier mentioned jukebox is an embedded device. This fact places constraints on the available processor speed, memory usage and energy consumption of the software. Researchers within Philips think that MySQL is not an optimal solution in the embedded environment. Especially an embedded environment needs to strike a good balance between constraints such as speed, energy usage and memory usage. In contrast, a modern PC always has enough energy and memory available for the tasks that are performed in an embedded environment. Although MySQL the RDBMS currently in use, MDB has a built in RDBMS abstraction layer that (ideally) allows replacing MySQL with another RDBMS without changing MDB. Therefore, Philips made the task of exploring the advantages and disadvantages of other ways to store the meta-data in an embedded environment, a graduation project. This report describes the process of finding the optimal solution(s).

1.2 Background

To put the problem statement from section 1.1 into a broader context, information has been gathered from internal Philips websites and technical notes.

About the organization:

"Storage devices derive their value to the user from their application." Within Philips Research, the group Storage Systems and Applications aims at innovations at the system level of storage devices to enable increasingly advanced applications of data storage, primarily in the home consumer environment.
Storage systems are considered in their context, which is determined by among others I/O devices, networks and data processors. Now that all information like audio and video has become digital, the multitude of options this creates for enhanced and new applications of storage is being identified and exploited.
Video recording is traditionally the most storage intensive application in the home and it naturally represents one of the main topics of the group. The recording, editing and management of compressed digital video in multiple streams on removable optical disks and fixed magnetic disks poses enormous challenges with respect to scheduling, defect management, data allocation and the like.
An explosive growth of available content combined with enormous increases in storage capacities is witnessed. More content means less value if the user cannot find its way through it. Data searching and retrieval mechanisms are studied in the group and integrated into the storage systems.
Future storage systems will all interact with networks, enabling among others distributed storage. This is also investigated in the group, both for the home environment and for mobile applications. Much of the output of the work finds its way into new ICs and software running on those ICs. System architecture and system integration are therefore key capabilities in the group.

About the PIC project:

The aim of the PIC project is to help Philips become a leader in Personal Infotainment. This should be achieved by integrating existing research results, new research results and emerging technologies into the PIC research prototype.
The PIC will allow the user to store all personally owned content, exchange this content with other users and enable local and remote playback and capture of content. Ease of use is facilitated by wireless networks and sophisticated user interfaces. Storage capacity is such that most applications do not fill the disk. A large A/V collection no longer fills a 300GB 1.8" HDD in 2007 [4]. Easy

[4] This project description is taken from [BvGS02] as published in 2002. Personal experience already contradicts the given numbers, but the ideas remain relevant.

(wireless) connectivity enables a compelling application: peer-to-peer content exchange. A good DRM solution protects the interests of the content industry and enables content exchange. Other features the device will contain are media processing technologies like layered (or scalable) encoding and content analysis features and application.

Chapter 2

Requirements

2.1 Introduction

In this chapter, concrete, measurable and verifiable requirements are distilled from the problem statement in chapter 1.

The first group of requirements (section 2.2) are requirements imposed by the developers steering the project. These are decisions based on cost, vision on the current and the future product line and technical limits.

The second group of requirements (section 2.3) are requirements set by (imaginary) future users of the PIC. These deal with the user perception of the PIC. These requirements are measured in terms of functionality, ease of use and number of features.

By combining all these requirements, several use-cases will be designed to serve as realistic examples of PIC usage. From these examples, the boundaries of the area where to look for alternative DBMSs will be defined.

2.2 Developer requirements

The first list with requirements is related to the MDB framework. They're beyond the scope of this report, but are mentioned in order to provide a complete picture of all requirements surrounding the project. The second list of requirements are specific requirements for the DBMS layer (the gray colored blocks in figure 2.1). They may be somewhat underspecified, but will be fleshed out in chapter 4. As will be seen later, conflicting requirements have to be balanced.

2.2.1 MDB framework related

- Management (adding, updating, deleting, searching and guarding consistency) of multimedia assets and asset collections.
- Provision of one API for multiple DBMSs.
- Support for overflow[1] meta-data.
- Support for meta-data interoperability (import and export of formats such as DIDL, MPV, MPEG7, MPEG21 and ID3.)

[1]Data of an unknown size, format or purpose. It's yet unknown in the design and implementation phase.

2.2.2 DBMS related

- *Extremely fast.* For delivering instant results to the user, certain important data access patterns should be really fast. This requirement is closely related to the energy efficiency requirement because faster, more optimized algorithms often require less energy.
- *Energy efficient.* Especially in embedded devices, all operations should require the least amount of energy to keep costs and battery size minimal.
- *Small memory footprint.* The storage solution should occupy a minimal amount of space to keep cost and size of memory hardware minimal. There is a trade-off between speed, capacity and costs. Smaller, fast memory such as DRAM is more expensive than cheaper, bigger memory such as a harddisk.
- *Scalable.* The chosen storage solution should work on a small embedded device, but also on a dedicated media server and any device in between. The system has to perform well in all cases.
- *Hardware platform independence.* Closely related to the scalability requirement. Because of the huge range of devices to support, it's good to keep the DBMS portable to allow adaptation to different architectures in the future.
- *Maintainable.* Software needs to be maintainable to guarantee correct operation in the future. Future programmers still need to be able to work on the software. This requirement is related to platform independence because a good abstraction level of the platform often means a cleaner DBMS architecture which makes it better maintainable. But also popularity among programmers is an indication of good maintainability.
- *Open source software.* Reasons for choosing open source software over proprietary solutions are the cost factor (it's free) and the availability of source code to improve it whenever there is a need to. Also, large community supported software projects tend to be well written, actively maintained and provide (mostly) free support. The disadvantage is required openness in produced software. Depending on the exact license of the software that is brought into the project, it may require any produced software to have the same license.
- *Linux based.* This is the mandatory platform. It is chosen by the project group because of its robustness, source availability, good platform support, productivity and low cost.

2.3 User requirements

While the mostly technical Philips requirements are important, the future users of the device also have requirements. It's hard to define what the typical user expectations of Philips devices are, because of the lack of information available on this subject. But there are some requirements that can be derived from 'common sense' and apply to most people:

- *Visual feedback.* There is a maximum amount of time that users are prepared to wait for the PIC to return results. Based on usability studies [Nie94], the response limit is set at 100ms. This is the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the results.

  Sometimes, an operation simply takes more time. Then it's good to show at least the relevant results within the time limit. Filling up the whole screen real-estate with anything that can be useful for the user is better than displaying a rotating hourglass, indicating progress. After displaying the initial results, more results can be retrieved[2], sorted or otherwise refined in

[2]Either by using an offset (although not in the SQL standard, it's possible in most DBMSs), or as a fallback method by executing the query again without or with a larger limit.

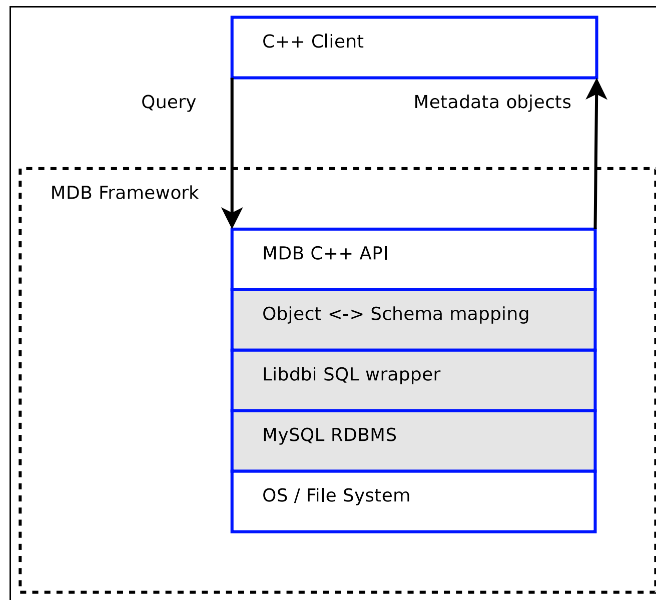

Figure 2.1: Components in the MDB framework.

the background. This lets the user have the impression of fast feedback while the completion of the operation is performed in the background.

- *Limited output.* It's unlikely the user wants to browse through huge lists of assets on the little display to find something. A limit can be defined for the amount of data to retrieve and display. Retrieving *all* data that is the result of a query is hardly ever what the user wants. The user is probably asking the wrong question and should be able to narrow the search. If and how support for "drill-down" queries should be implemented is a topic that has to be solved first on the UI level. This is beyond the scope of this report.

- *Sorting.* The user expects a certain structure in the output. Alphabetical sorting of data is often used, but may be resource intensive. But this may not always be what the user wants. It may be better to sort on relevance. This can be as simple as ranking the most frequently used files as more relevant than less frequently used files. A more advanced example: the Google[3] search-engine also sorts on relevance using their PageRank technology[4] to deliver results to the user.

2.4 Current Situation

For a better understanding of the extended assignment, more detailed knowledge on MDB is required. In figure 2.1, the MDB framework is expressed as a diagram of layers which will be explained in detail below.

The **MDB C++ API**[5] component provides a set of routines, defined objects and relationships between objects for building software applications. A good API makes it easier to develop an application by providing all the building blocks. The application programmer puts the blocks together.

[3] `http://www.google.com`
[4] More details at: `http://www.google.com/intl/en/technology/`
[5] Application Program Interface

The **C++ Client** component, like the jukebox GUI introduced in section 1.1 is such an application using the building blocks provided by the API. Several routines offered by the MDB API provide the ability to store, modify, search and retrieve assets.
Recall from section 1.1 that an asset is an object containing all sorts of information about a multimedia file. The object, like an object in C++, has properties (artist, track number, album name, etc.) and methods (mainly property getters and setters). Below this layer, the design has been built bottom up. RDBMS is a proven technology. Therefore the popular MySQL RDBMS was chosen as storage backend. Because MySQL is a relational DBMS, not an object-oriented DBMS, these assets can't be directly stored in MySQL. As a consequence, the **Object to schema mapping** component extracts the properties of an asset so they fit into the schema of different tables in a relational database. The reverse is also performed by the component: assembling an asset by retrieving records from the related tables[6].

The Libdbi SQL wrapper is the current instantiation of the RDBMS abstraction layer that was introduced in section 1.1. This component, which is also an API, provides the essential routines for establishing a connection to an RDBMS and for performing SQL compatible queries. The layers above the Libdbi layer in figure 2.1 use this *one* set of routines provided by Libdbi to communicate to any RDBMS. The layer directly beneath Libdbi is the RDBMS driven by Libdbi. In this particular case, it's **MySQL**, but it may also be another RDBMS, or multiple RDBMSs as can be seen in B.3.

The **OS and File System** are currently Linux and Ext2. Linux because it is the mandatory OS, as listed in the requirements. Ext2 because it's the default File System for most Linux installations[7].

2.5 Two sub-projects

In the original assignment, a research project was formulated by Philips Research:

Philips Research is working on a universal platform or framework called MDB that could be the basis of storage software for each future Philips device that contains multimedia data. Performance is not satisfactory, the framework is operating too slow. They have two questions regarding the so called research DB:

- In which layers of figure 2.1 are the biggest speed bottlenecks?
- Is there a "better" storage solution than MySQL with Libdbi?

A few months after this graduation project started, another project was introduced:

Philips builds MP3 players. The HDD060 type is currently sold in stores[8]. It features a 1.5GB harddisk. Internally, it uses the SQLite RDBMS. On top, a small, proprietary C++ API is used for interfacing the RDBMS with the client application. For the next generation of audio jukeboxes, they have one question about the so called product DB:

- Is there a better storage solution than SQLite?

[6] As will be seen in section 3.3.5, the conversion process can be automated
[7] The combination of OS and file system can greatly influence energy usage [Mes03], but this is below the scope of this project.
[8] `http://www.p4c.philips.com/files/h/hdd060_00/hdd060_00_dfu_eng.pdf`

The goals of the two projects are not identical, but do overlap. This overlap has to be identified to work efficiently towards answering the three questions.
While it makes sense to first figure out where the speed bottleneck is and then see if it's still worthwhile to find a better DBMS, this approach is not chosen for three reasons:

- Finding out the bottleneck was an item that came up in the first weeks of the project after brainstorming about the possible approaches to create a better framework. My original assignment was DBMS related, which should be my main graduation subject. Changing this to anything non-DBMS related would be far out of scope. Therefore, finding the bottleneck is now put on a side track. The emphasis of this report is now on DBMSs.
- Answering the DBMS related questions of the two projects also requires an emphasis on DBMSs. It takes more time than answering the bottleneck question.
- Philips expects more value from the answers to the DBMS question.

The best way to proceed on answering the DBMS related questions is discussed in the next few sections.

2.6 Hardware target platforms

The answer to the question of what the "best" DBMS is, depends on the definition of the "best". Project "research DB" needs the "best" DBMS in a broad range of products while a DBMS for project "product DB" needs only to be very good in a small range of products. To formulate the final definition of the "best" DBMS, the product ranges need to be specified first. Therefore, the product range is divided into three target platforms. The hardware descriptions are based on the future vision of Philips. Each target platform has an architecture and a typical set of use-cases. The use-cases are defined in section 2.8.

2.6.1 Target platform 1: mobile audio player

This is the low-end device from the range of devices of the "research DB". This is also the only target device of the "product DB". This target is primarily used as a single user system. Contents on this target are exclusively audio. The schema that the MDB framework uses on this target is figure D.2. Philips Research currently uses Melody hardware as a prototype.

The hardware specifications for this platform:

- Melody/SSA 133 MHz processor
- 32MB RAM
- 3GB harddisk

2.6.2 Target platform 2: mobile audio/video player

This is a target device of the "research DB". It's primarily used as single user system. Contents on this target are audio, video and pictures. The schema that the MDB framework uses on this target is figure D.3. Philips Research currently uses XSilo hardware as a prototype.

The hardware specifications for this platform:

- Intel XScale 400 MHz processor
- 64MB RAM
- 40GB harddisk

2.6.3 Target platform 3: home media server

This is the high-end target device of the "research DB". It's primarily used as multi user system. Contents on this target are audio, video and pictures. It has also knowledge of users and rights. The schema that the MDB framework uses on this target is figure D.4. Philips Research currently uses a standard desktop platform as a prototype.

The hardware specifications for this platform:

- Intel Pentium 4 2.4GHz processor
- 1GB RAM
- 300GB harddisk

2.7 Scenarios

To provide an accurate answer for both projects at the same time, the DBMS questions are refined using 4 scenarios:

- What is the "best" DBMS for target platform 1.
- What is the "best" DBMS for target platform 2.
- What is the "best" DBMS for target platform 3.
- What is the "best" DBMS for all three target platforms.

The "best" DBMS can be defined as:

- The DBMS that meets the most important requirements for the given target platform(s).

2.8 Use-cases

To add a measurable dimension to the definition of "best", several use-cases are defined in this section. These use-cases are common situations of PIC usage. Every use-case is rated on importance with regard to the target platform. The higher the weight for a use-case is, the more important the result of the use-case is and the more it influences the outcome. Per platform, a 100% score is to be devided among the use-cases so it'll indicate which use-cases are especially important on what platform. The scores are also added to indicate which use-cases are important to all platforms combined.

1. Joe bought a brand new PIC. He wants to copy all his music from his laptop to his PIC.
2. John wants a quick overview of all music on his PIC. He just wants a list of songs.
3. Janet wants to search directly for a song title and play it.
4. Jill wants to see a list of available albums on her PIC. After selecting an album, she wants to see what songs are in it.
5. Jasper has a huge collection of music on his PIC. He wants a structured, sorted overview of all his music. He first wants to select an artist, then the album, then the song to play.

Use-case	Target 1 (weight)	Target 2 (weight)	Target 3 (weight)	All targets (weight)
1 (Joe)	5	5	20	30
2 (John)	10	10	40	60
3 (Janet)	40	40	5	85
4 (Jill)	40	40	5	85
5 (Jasper)	5	5	30	40
	100	**100**	**100**	**300**

Table 2.1: Frequency table of use-cases for each target platform.

The "best" DBMS for each target is defined as:

$$R_{tot} = \sum_{i=1}^{5}(W_i R_i) \tag{2.1}$$

Where W_i is the weight of use-case i, R_i is the result of the benchmark for use-case i in seconds and R_{tot} is the total result for the DBMS in seconds. The total weight to be divided per target is 100. This formula is a tool to help appointing the right DBMS for the platforms.

To skip directly to the results, start reading from chapter 6 where the DBMSs will be evaluated according to table 2.1 and formula 2.1 using the benchmark measurements.

Note that the numbers in table 2.1 are based on my personal experience with a portable media device. What this document provides is a method for DBMS performance scoring. As the use-cases and weights will change over time, these should be looked upon as an example, although currently accurate and useful.

To learn more about the background and different database related concepts, read chapter 3 first.

Chapter 3

Basic concepts

3.1 Introduction

In this chapter, the existing storage techniques and systems are explored and compared. The area has to be mapped first in order to perform benchmarks on the found storage systems.

For the DBMSs that will be benchmarked later on, detailed information about the simple scan and a join operations is provided. The concept of caching (in relation to databases) is also briefly introduced in section 3.4 to help interpreting the database measurement results in chapters 6 and 7.

As some of these concepts are considered basic knowledge in the database field, feel free to skip them.

But first, the concept of meta-data is explained in section 3.2. It's important to know the properties of the data that is going to be stored in the database that is benchmarked throughout the project.

3.2 Meta-data

Imagine a large collection of multimedia files. How does one search within this collection? There's only one way the files can be identified (and thus sorted and searched for): by file name. A file name can represent some information about the multimedia file, but it's limited to (a certain amount of) text. Every file would ideally have a short "*summary*" of the data it contains so that searching on all properties is possible. Such a summary is called meta-data[1]: data about data.

Meta-data can be stored within the file itself, or in a dedicated store (then there should remain some kind of link to the file). Both methods have their usage domain. A comparison:

- Speed: when searching often, or on high latency (remote) file systems, a dedicated store definitely speeds up the searches.
- Simplicity: keeping meta-data inside a file allows the file to be moved around without the dedicated store dependency that would otherwise require updating on the source and destination machine.
- Maintenance: keeping meta-data inside a file won't require the setup and maintenance of a dedicated store.

[1] `http://en.wikipedia.org/wiki/Metadata_(computing)`

Nowadays, meta-data is still stored in the data files themselves. But there's increased demand for dedicated stores, because of audio fingerprinting [JHB02] for example: a tool extracts unique features from an audio track and transforms these into a large (integer) number that matches only this particular track. This number in sent to a dedicated store (typically through the Internet[2]) and meta-data is retrieved. The meta-data is often continuously updated by companies and individuals around the world. Popular among audio files is a hybrid use: keeping meta-data within the file, but synchronizing updates with a dedicate store (in both directions). This hybrid form has all of the advantages mentioned earlier, but unfortunately the meta-data redundancy requires extra space and needs to keep both the meta-data in the file and the dedicated store consistent.

3.2.1 MPEG meta-data example

To quantify how much space is occupied by using a dedicated store to keep the meta-data in, an MPEG audio file is dissected:

An MPEG audio file is built of small parts called frames. Each frame has its own header and audio information[3].
It's usually enough to find the first frame, read its header and assume that the other frames are the same. This may not be always the case. Variable bit-rate MPEG files may use so called bit-rate switching, which means that the bit-rate changes according to the content of each frame.

The frame header is constituted by the very first four bytes (32bits) in a frame. The first eleven bits of a frame header are always set and they are called "frame sync".
Then there's the non-technical meta-data: the ID3 tag. Most modern MPEG audio files include both the ID3v1 and ID3v2 tag. While ID3v1 is limited in the variety of meta-data it is capable to describe, it's there for compatibility with older audio players. It contains 125 bytes of meta-data. There is no default size of an ID3v2 tag. It may even contain embedded images[4].

The estimated total amount of meta-data per audio file is 4B (header) +125B (ID3v1) + 2048B (ID3v2 size guess) = 2177B.

Using the size of a random, regular MPEG file, of 7992KB, the ratio estimate is 3600:1 (7992000:2177). Then target 1 (the mobile 3GB audio player) would hold around 830KB of meta-data. Target 3, (the 300GB home server) would hold about 83MB of meta-data. To put these numbers into context, a couple of side notes have to be made:

- For all types of media files, there *can* be a lot more meta-data to store. A piece of software may generate meta-data for each video frame in a movie, or determine the instruments used in an audio file.
- When data is stored in a database, it is always accompanied with a certain amount of space overhead. For example, when using an XML database to store the name of the artist, tags will be placed around the name. This may cost a lot of extra space.

3.3 Database concepts

To explore the idea of a dedicated store mentioned in section 3.2 further, this section provides an overview of several methods for storing and searching data. The terminology introduced in this section is similar as in [EN00], which will be used throughout this document.

[2]Such a service is for example: `http://www.freedb.org`
[3]More details at: `http://www.id3.org/mp3frame.html`
[4]More details at: `http://www.id3.org/intro.html`

Data are known facts that have implicit meaning. The word *database* is in such common use that is has to be defined first. A **database** is a collection of related data and has the following properties:

- A database is a logically coherent collection of data.
- A database is designed, built and populated with data for a specific purpose.
- A database can be of any size and of varying complexity.

A database, such as a card catalog used in libraries, is generated and maintained manually. If these tasks are performed by a collection of programs then this computerized solution is called a **database management system (DBMS)**. A DBMS is a software system that facilitates the following processes:

- *Defining* a database by specifying types, structures and constraints for the data to be stored in the database.
- *Constructing* a database by storing the data itself on some storage medium that is controlled by the DBMS.
- *Manipulating* a database by querying the database to retrieve or update specific data.

Together, the database and the DBMS software are called a **database system**.

Although the choice has already been made for the author, the implications of using a database or DBMS versus a file with data are listed below:

- Once a database is up and running, creating a new application using it is cheap.
- Better information availability for multiple users through concurrency control and recovery subsystems of a DBMS.
- Overhead costs of features like security, concurrency control, recovery and integrity functions that are typically provided in a DBMS by default.

By looking at the requirements from section 2.2, where speed is the most important consideration, using a DBMS doesn't seem like a good idea. Unfortunately, using expensive features such as concurrency control is really a necessity. As explained in section 2.6, a single solution needs to cater for a broad range of devices, including a multi-user server. Therefore, several types of conventional and research prototype DBMSs are reviewed in the remainder of this section.

3.3.1 Tree data structures

Tree data structures are often used as a simple way to store data. There are algorithms available that allow efficient operations on the data in the tree structure. This is tedious work when the collection of data to be managed becomes complex. Therefore, many database systems incorporate these tree data structures (like BDB, see section 3.3.2) but don't expose them to users directly (like RDBMSs, OODBMSs and O/RMSs in sections 3.3.3 to 3.3.5).

Skipping the flat file database model[5], a short introduction and overview of tree data structures will now follow:

A tree structure is a computer data structure[6] that emulates a natural tree structure with a set of linked nodes. See figure 3.1 for an example.

[5]Details at: http://en.wikipedia.org/wiki/Flat_file_database
[6]Details at: http://en.wikipedia.org/wiki/Tree_data_structure

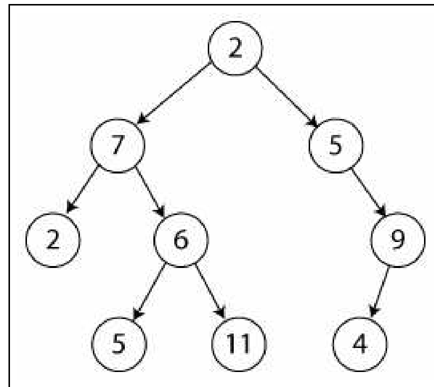

Figure 3.1: A tree structure. Circles represent nodes. Numbers represent keys.

Each node has zero or more child nodes. A node that has a child is called the child's parent node. A child has exactly one parent; a node without a parent is called the root node (or root). Nodes with no children are called leaf nodes.

Trees are either unordered or ordered. Ordered trees are easily searchable and therefore the most interesting type.

A **BST** (binary search tree) is a binary tree where every node's left sub-tree has keys with values less than the node's key value, and every right sub-tree has keys with values greater than the node's key value. There are many types of BSTs. AVL trees and red-black trees are both forms of self-balancing binary search trees. A splay tree[7] is a BST that automatically moves frequently accessed elements nearer to the root (that's where the searching starts).

The trees mentioned in the remainder of this subsection are becoming less general and often specialize in solving problems in a certain area, in an attempt to save time or space.

B-trees[8] have substantial advantages over alternative implementations when node access times far exceed access times within nodes. This usually occurs when most nodes are in secondary storage such as hard drives. By increasing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often, and efficiency increases. Usually this value is set such that each node takes up a full disk block or comparable volume in secondary storage.

The **B+-tree** is a variation on the B-tree. In contrast to a B-tree, all data are saved in the leaves. Internal nodes contain only keys and tree pointers. All leaves are at the same lowest level. Leaf nodes are also linked together as a linked list to make range queries easy.

The **B*-tree** generally keeps each node fuller resulting in trees that are more shallow and thus leading to faster searches. Insertions are more expensive as a consequence.

The **dancing tree** is a tree data structure that is used by Namesys' Reiser4[9] file system. As opposed to self-balancing binary search trees that attempt to keep their nodes balanced at all times, dancing trees only balance their nodes when flushing data to a disk. Because writing to disk is much slower than writing to memory, this may be a benefit in some areas.

Implementations

A tree data structure in itself doesn't make a DBMS, but this knowledge is invaluable to better understand and use DBMSs that are yet to be described. The implementation that tries the least to cover the pure tree data structure under a syntax sugar coating is **BDB** (Berkeley DB). BDB provides developers with DBMS features such as concurrency control but still allowing choice in

[7]Good for implementing caches. More details at: `http://en.wikipedia.org/wiki/Splay_tree`
[8]More details at: `http://en.wikipedia.org/wiki/Btree`
[9]More details at:`http://www.namesys.com/v4/v4.html`

low-level matters such as choosing an indexing mechanism that fits the application best. As stated earlier, there's no single best solution in every situation.

Because BDB proves to be a good solution in the measurements from chapter 6, it's worthwhile to provide in-depth information about BDB. Section 3.3.2 is dedicated to the workings of BDB.

3.3.2 Berkeley DB

Introduction

Information about popular RDBMSs is easily found on the Internet. Information about Berkeley DB (BDB) is much less common, therefore this section tries to fill the gap.

Knowledge about tree data structures (see section 3.3.1) is a prerequisite to understand the remainder of this section. Make sure to read it first.

First in this section, Berkeley DB (BDB) is introduced. Next, some practical information is given about searching and joining in BDB. Joining in BDB is a more difficult technique compared to the SQL join as used in RDBMSs because programmers need to implement it themselves. Naturally, once a database operation is programmed as a function, it can be reused elsewhere in the application.

BDB, the self proclaimed most widely-used developer database is open source and runs on all major operating systems. These are the distinctive BDB features:

- Small footprint (less than 500Kb)
- Simplicity of integration into an application, no need for an external DBMS.
- High performance
- Supports transactions, so that multiple changes can be applied or rolled back atomically.

BDB has these critical disadvantages:

- Steep learning curve for the programmer.
- Reduced data portability. Transferring data from one RDBMS to another is ”easy” using SQL. But no other DBMS interfaces with BDB in this way.
- Programming applications is a more complex and time consuming task compared to RDBMSs.

As will become clear in the following sections, an application that makes intensive use of the database will require a lot of programming. Using BDB is only recommended when the benefits outweigh the disadvantages by far. BDB is very suitable in applications with static schemas and a small static set of queries.

Simple scan

In figure 3.2, table ”Albums (primary)” contains all available data on albums. This is called the primary table in BDB lingo. There's only one key field in such a BDB table. This key field can be searched on. An example with search terms is given in section 3.3.2. If no search conditions are specified, the search has become a simple scan that retrieves all records in the table.

Sample code:

```
// Instantiate a new database object.
// In BDB, a set of key-value pairs is called a database.
// In an RDBMS it would be the equivalent of a table.
albumDB = new MyDb(albumDBDirName, albumDBFileName);   // Database location

// Create a cursor in order to perform queries
Dbc * cursorp;
try {
        albumDB->getDb().cursor(NULL, &cursorp, 0);

        // Iterate from the first matching record to the last, displaying each
        Dbt key, data;
        while ((ret = cursorp->get(&key, &data, DB_NEXT)) == 0) {
                AlbumData albumItem(data.get_data());
                // Do something with the data
                albumItem.show();
        }
} catch(DbException & e) {
        // Handle any errors
        albumDB->getDb().err(e.get_errno(), "Oops!");
        cursorp->close();
        return 1;
}
// Clean up after having used the cursor
cursorp->close();
```

Advanced search

Sometimes, other fields than the key field need to be searched. To provide alternative keys to search on, one or more additional tables have to be created: the so-called secondary tables. They're always derived from the primary table. A secondary table can be considered an index on the primary table. The index can be made of various trees or a hash table. The worst-case performance on a search could be *O(log n)*, where n is the amount of records in the table. Not having these secondary tables still allows one to search on other fields, but this means scanning through every single record in the primary table in a brute-force manner. In the worst-case scenario, this is an exhaustive search of *O(n)*.

Whether to create secondary tables is a trade-off between desired search speed and additional space requirements for the secondary tables. For target platforms 1 and 2 (defined in section 2.6), the processor is a more critical resource than for target platform 3. More on the speed and amount of storage of all target platforms in section 3.4.

Secondary tables are always managed by BDB. To have BDB create a secondary table, it needs to be associated with the primary table and a callback needs to be defined for this secondary table. The input for this callback is a data record from the primary table, the output is the key value for the secondary table. As an interesting side note: keys on primary table should of course be unique, but on secondary tables, keys don't have to be unique. This feature can both be seen as an advantage (allows a simple key extractor) and a disadvantage (a query could include unwanted duplicate records more easily).

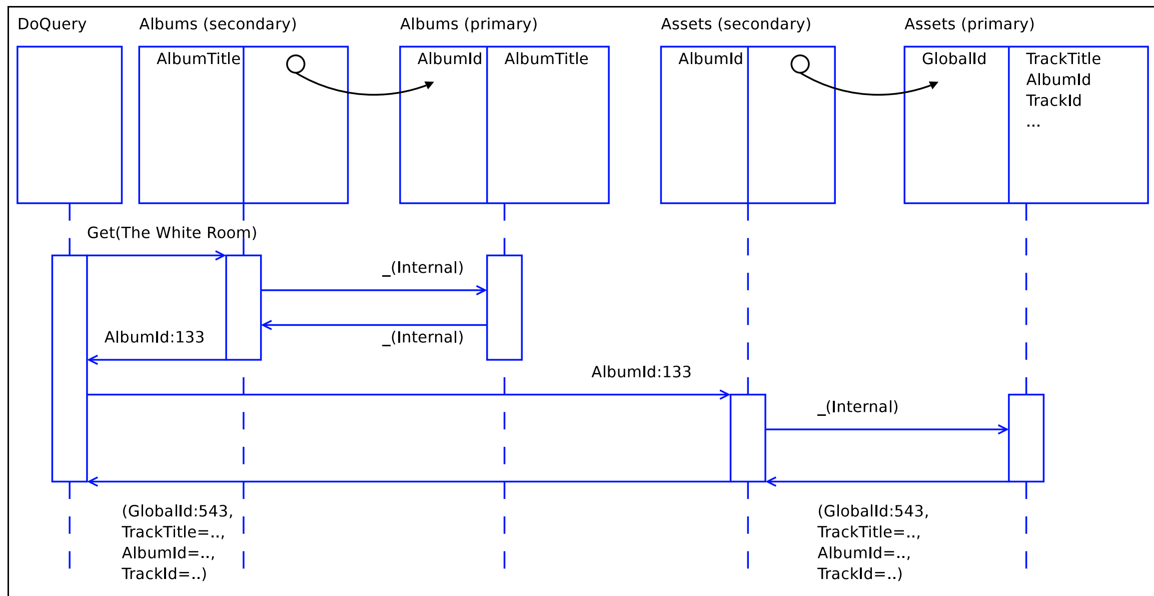

Figure 3.2: Sequence diagram of two sequential searches in BDB. The query: retrieve the details on all albums called "The White Room".

The following code example is the callback as it is used for creating the secondary Albums table in sequence diagram 3.2:

```
int albumtitle_callback(Db * dbp,// The primary DB handle
               const Dbt * pkey,        // The primary DB record's key (AlbumId)
               const Dbt * pdata,       // The primary DB record's data (AlbumTitle)
               Dbt * skey)   // The to be created secondary DB record key
{
// Extract the data from the record
        AlbumData id(pdata->get_data());
// Extract the AlbumTitle from the data
        const char *itemname = id.gettitle().c_str();
// Assign the AlbumTitle as the new key for the secondary DB
        skey->set_data((void *) itemname);
        skey->set_size(strlen(itemname) + 1);
        // Return success, BDB will automatically add this new key to the secondary DB
        return 0;
}
```

An example of a search using a secondary table is visualized in diagram 3.2.

Join

An equivalent of joining tables as it exists in an RDBMS doesn't exist in BDB. BDB can only join multiple secondary tables to perform a search, never any primary tables. Step 2 in sequence diagram 3.3 shows how records from the primary table "Assets" are requested that match a certain TrackTitle and TrackId. The match is made using two secondary tables simultaneously in a join.

To simulate a join as in an RDBMS, Step 1 has to be carried out before step 2. In step 1, AlbumIds are retrieved from the records in the primary "Albums" table that match certain search terms. By performing step 2 for every record found by step 1, a join between two primary tables is performed.

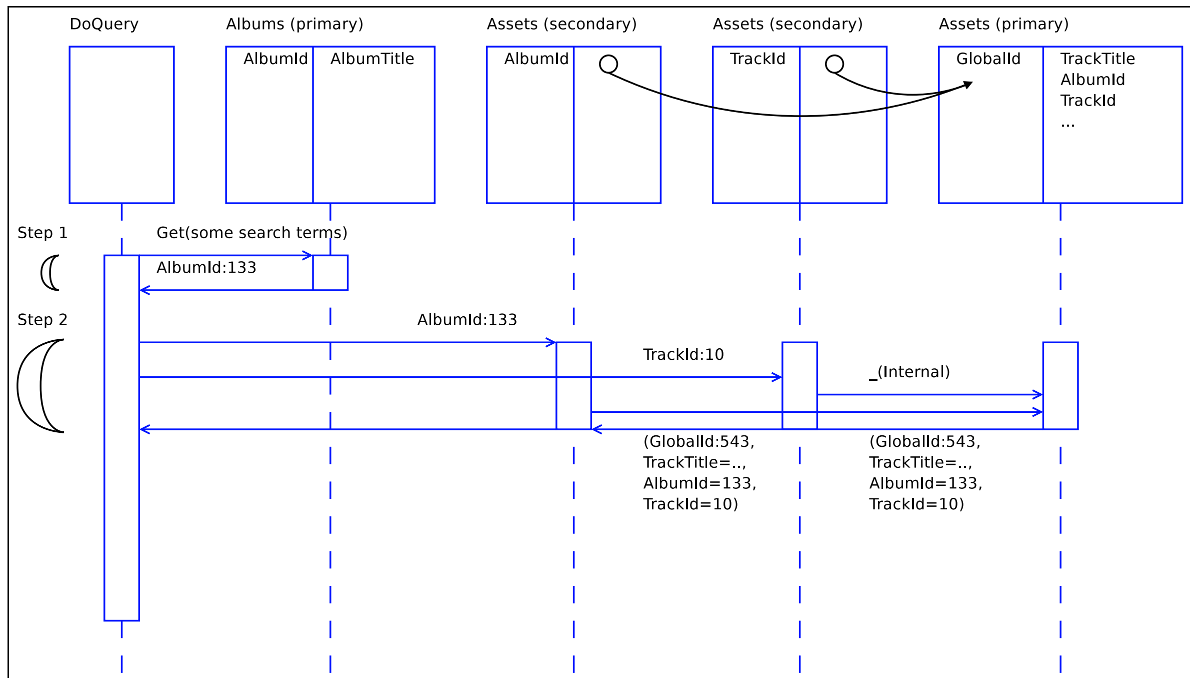

Figure 3.3: Sequence diagram of a search (step 1) and a join (step 2) in BDB. The query in step 2: retrieve the details on all albums matching an AlbumTitle expression and contain at least 10 tracks.

A code example for this type query is rather complex and can be found in the framework in testcase 6.

More information on BDB is best read in the BDB C++ tutorial at the BDB website: `http://www.sleepycat.com/docs/gsg/CXX/BerkeleyDB-Core-Cxx-GSG.pdf`.

3.3.3 RDBMS

An RDBMS (Relational DataBase Management System) is a DBMS that is based on the relational model. In this model, all data are represented as mathematical relations. Reasoning about such data is done in predicate logic. The basic relational building block is the domain. A tuple is a set of ordered pairs of domain and value. A relation is a set of tuples. Comparing these definitions to traditional database concepts, a table is the visual representation of a relation and a tuple would resemble a row.

The advantages:

- If one value is dependent on another, this dependency is enforced through referential integrity.
- Any column in a relational database can be searched for values.
- Support for ACID transactions[10]
- Support for very large databases
- Searching without conventional programming with SQL (Structured English Query Language)

[10] Atomic, Consistent, Isolated and Durable.

- Automatic optimization of searching
- Indexes can be used to improve search efficiency

The disadvantages:

- Not everything can be indexed (partial matching of text in text fields is inefficient)
- Poor support for storage of complex objects (video, images)
- Joins (retrieving cross-table information) are expensive in terms of processing time.

Implementations

Many popular free and commercial implementations exist and are actively used. An up-to-date list of RDBMSs with comparisons is available on the Internet[11].

- MySQL (free)
- PostgreSQL (free)
- SQLite (free)
- Firebird (free)
- DB2 (commercial)
- Oracle (commercial)
- mSQL (commercial)
- Ingres (commercial)

The inner workings of the queries used in the benchmarks described in the later chapters will be discussed in the following subsections. Unlike the direct, deterministic scan and join in BDB from section 3.3.2, RDBMSs are instructed with a less restrictive query that first has to pass the query optimizer before execution. Therefore, the query optimizer is introduced first, followed by the details on the simple scan and join implementations.

Query optimizer

Query performance in RDBMSs is not only dependent on the database structure, but also on the way in which the query is optimized. SQL is the widely used query language for RDBMSs. SQL query processing requires that the DBMSs identifies and executes a strategy for retrieving the results of a query[12]. The SQL determines what data is to be found, but does not define the method by which the data manager searches the database. Four steps are necessary for processing an SQL query. They'll be explained by example in the next subsection.

[11] http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

[12] More details can be found in the following paper at: http://www.kisekaeworld.com/Intractable/SQLOPT.pdf

Simple scan

To illustrate how a simple scan SQL query is processed by a RDBMS, the four necessary steps will now be described.

The first step in the process is parsing the query. Three parts can be separated by syntactic analysis: SELECT x FROM y WHERE z.

In the next step, the query optimizer has the task of determining the optimal execution plan for this SQL query. A simple approach is to use a rule based method for ordering the operations in a query execution plan. A first rule could be to use an index[13] if available, instead of performing a full table scan. Assuming a balanced tree based index, a scan trough the index will take $log_2(n)$ accesses in the worst case. Contrast this with performing a full table scan would take $n/2$ access on average or n accesses if the entry is not existing.

Even though query optimizer are quite advanced, quite often, there's no optimal strategy found because there's simply not enough information available about the data. This problem is best illustrated by introducing the concept of *selectivity*. Selectivity is defined as the ratio between the number of rows that satisfy a condition to the total number of rows in the table (while assuming a uniform distribution). Say for example, there are two WHERE clauses (genre="trance" AND artist="madonna") that both have to be satisfied in order to include a row in the results. If the database is mostly filled with music from madonna (say, every 9 out of 10 tracks in the table) which is not trance except for a few remixes (say, every 2 out of 10 tracks is trance), it's faster to select the rows that satisfy genre="trance" first, and then check if those rows also match artist="madonna" instead of the other way around. The selectivity on artist is $9/10 = 0.9$ and on genre $2/10 = 0.2$. When the database holds 500 tracks, the slowest search plan accesses $500 * 0.9 = 450$ genre tests versus $500 * 0.2 = 100$ artist checks in the fast search plan. The problem is that the DBMS often doesn't contain any knowledge on the selectivity and is not able to make the optimal choice. In these cases, the order in which the clauses are specified in the SQL query is usually taken as a hint for the query optimizer.

In the third step, the query code generator is activated. The query execution plan from the second step is taken as input and will now be converted to code that is executable by the run-time database processor.

In the fourth and last step, the run-time database processor executes the code output from the third step and returns the results. Depending on the complexity of the DBMS, more steps may be taken. If the DBMS supports concurrency, the run-time database processor sends requests to the transaction manager to make sure no other request conflicts. Then there's the storage manager to control the (mapping of) records in memory or disk. There can be more managers involved. But as always, more managers make situations slower and more complex. Therefore, if these managers are not required by the application, they can be turned off in some DBMSs.

Join

One common technique for joining is a nested loop. It works like the method described in the previous subsection. Depending on the join selectivity factor, the table with the least matches is appointed for the outer loop. The optimal join takes into account the table sizes and the computed estimated join selectivities.

Another technique for joining is the merge join. The query optimizer would choose this type of join only if both tables have already been sorted on the join column. Each table is only traversed once.

Concluding, it's important to understand and work with the query optimizer to gain optimal results by carefully constructing SQL queries.

[13]Indexes are explained in subsection 3.3.2

3.3.4 OODBMS

An OODBMS (Object Oriented DataBase Management System) is a DBMS that stores objects, as opposed to tuples or records in an RDBMS. Because data is stored as objects it can be interpreted only by using the methods, usually specified by its class. The relationship between similar objects is preserved (inheritance) as are references between objects. An OODBMS is usually associated to a object-oriented programming environment.

The advantages:

- Queries can be faster because joins are not often needed. This is because an object can be retrieved directly without a search, by following its object id
- The same programming language can be used for both data definition and data manipulation
- OODBMSs typically provide better support for versioning

The disadvantages:

- OODBMS isn't a mature and proven technology like the SQL DBMSs that have had decades of development and testing
- Still missing tools and features like industry standard connectivity, reporting tools, backup and recovery standards
- OODBMS lacks a formal mathematical foundation, unlike the relational model, which rests upon the firm and well-understood mathematical basis of the relational calculus

Implementations

Some free and commercial implementations exist, but are not widely used. An up-to-date list of OODBMSs is available on the Internet[14].

- db4o (free)
- GOODS (free)
- Caché (commercial)
- ObjectStore (commercial)
- Objectivity (commercial)
- Versant (commercial)
- JADE (commercial)

Before diving into the detailed query operations, a brief introduction on the object model is in order.

Upon the OMDB object model, the object definition language (ODL) and object query language (OQL) are based. This model provides data types and other concepts that are used in the ODL to specify the object database schema. The ODL is not explained in depth here as it is similar to the familiar constructions in OO programming languages. Except for properties and operations, classes can also have relationships to other classes. A relationship between two classes is defined by two references to each other.

The OQL is explained by example in the next subsection.

[14] `http://en.wikipedia.org/wiki/OODBMS`

Simple scan

The basic OQL syntax is similar as the SQL syntax from subsection 3.3.3. A selection is made by specifying an entry point, a set of names of persistent objects for example. The conditions are specified in the WHERE clause, a property of a certain value for example. Then the result is specified, a certain field for each matched object for example.

The result consists of a bag of path expressions. A path expression can be an entry point or an iterator[15] variable over a range of objects.

Join

Although the concept of a join in OQL is similar as the SQL join in many ways, it's more difficult to say anything about the run-time. According to [Wan00], query evaluation can be based on either the relational data model or higher-ordered data models (that are beyond the scope of this document). But there are several issues that exist in a object context. Joins in OQL are joins between multi-sorted collections. This means that collections in the FROM clause are not always n-arity relations as with SQL, but can contain collections of objects or collections of collections.

Where SQL imposes a relatively simple database structure, OQL allows much more freedom which makes (run-time) analysis substantially harder to do.

3.3.5 O/RMS

An Object Relational Mapping System is a programming technique that links relational databases to object-oriented language concepts, creating (in effect) a "virtual object database." When applications interface to this type of database, it will normally interface as if the data is stored as objects. However the system will convert the object information into data tables with rows and columns and handle the data the same as a relational database. Likewise, when the data is retrieved, it must be reassembled from simple data into complex objects.

The ability to directly manipulate data stored in a relational database using an object programming language is called transparent persistence. This is in contrast to an interface used by ODBC[16] or a custom API. With transparent persistence, the manipulation and traversal of persistent objects is performed directly by the object programming language in the same manner as in-memory, non-persistent objects.

The advantages:

- The software to convert the object data between an RDBMS format and object database format is provided. Therefore it is not necessary for programmers to write code to convert between the two formats and database access is easy from an object oriented computer language.

The disadvantages:

- Because the O/RMS converts data between an object oriented format and RDBMS format, performance of the database is degraded substantially. This is due to the additional conversion work the system must perform.

[15] An iterator is an object allowing one to sequence through all of the elements or parts contained in some other object, typically a container or list.

[16] ODBC: Open DataBase Connectivity, a standard database access method.

Implementations

Some free and commercial implementations exist. An up-to-date list of frameworks and mapping tools is available on the Internet[17].

- Postgres (commercial)
- PostgreSQL (free)

When choosing whether to use a O/RMS or another DBMS type, the trade-off between developer ease of use and speed has to be decided upon.

3.3.6 XML DBMS

The XML database topic is quite broad. Only some aspects will be highlighted.

An XML[18] document is a database only in the strictest sense of the term: it is a collection of data. As a database format, XML has some advantages. For example, it is self-describing (the markup describes the structure and type names of the data, although not the semantics), it is portable (Unicode), and it can describe data in tree or graph structures. It also has some disadvantages. For example, it is verbose and access to the data is slow due to parsing and text conversion.

Through extensions, XML provides many features found in DBMSs: storage (XML documents), schemas (DTDs, XML Schemas, RELAX NG), query languages (XQuery, XPath), programming interfaces (SAX, DOM), and so on.

But it's lacking efficient storage, indexes, security, transactions and data integrity and multi-user access.

As a solution for most of these concerns, native XML databases exist. They fall into two broad categories:

- **Text-based storage**. Store the entire document in text form and provide some sort of database functionality in accessing the document. A simple strategy for this might store the document as a BLOB in a relational database or as a file in a file system and provide XML-aware indexes over the document. A more sophisticated strategy might store the document in a custom, optimized data store with indexes, transaction support, and so on.
- **Model-based storage**. Store a binary model of the document (such as the DOM[19] or a variant thereof) in an existing or custom data store. For example, this might map the DOM to relational tables such as Elements, Attributes, Entities or store the DOM in pre-parsed form in a data store written specifically for this task.

The XML DBMS field is not mature yet. There are not many implementations and what's implemented is often an XML layer on top of an existing RDBMS. A cloud of dust surrounds the definition "native" XML DBMS. Some vendors define native as storing XML data in an underlying storage mechanism that should map directly on the XML document, not re-using an RDBMS that is build with tables and relations.

Implementations Some free and commercial implementations exist:

- 4Suite (free, library)
- Berkeley DB XML (free, "native", embedded)
- X-Hive (commercial, native, server)

[17] `http://en.wikipedia.org/wiki/Object-relational_database`

[18] The Extensible Markup Language is a W3C-recommended general-purpose markup language for creating special-purpose markup languages.

[19] Document Object Model, a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.

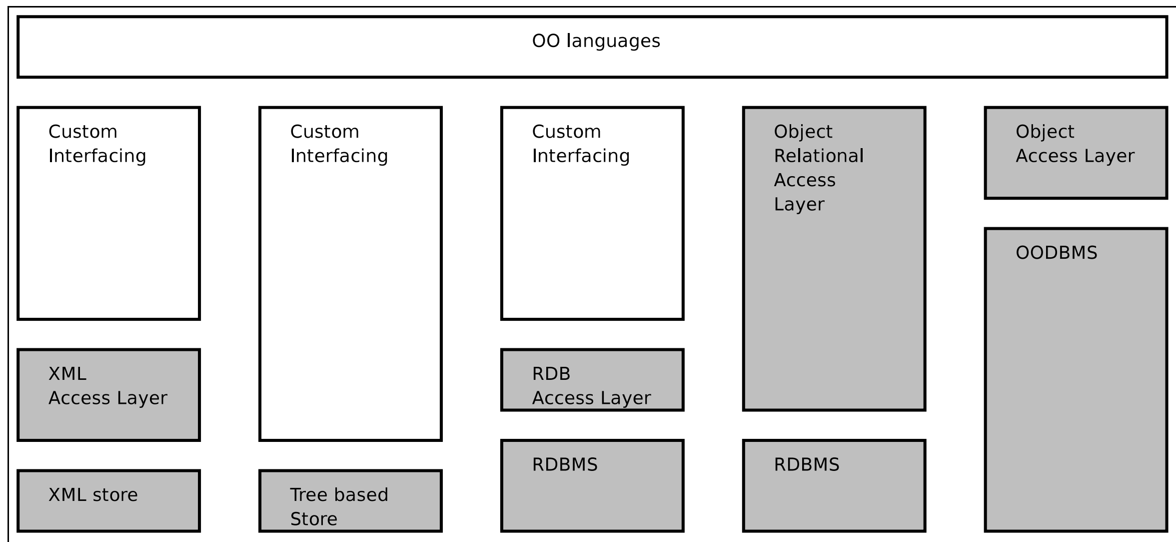

Figure 3.4: Custom and provided layers per type of store.

3.3.7 DBMS summary

How all databases described in the previous subsections would replace the current MDB DBMS (or any other OO based application) is visualized in figure 3.4. An XML DBMS won't be fast and mature enough to be proposed as a serious candidate. Although XML might have an advantage over other DBMSs when it comes to interoperability. An XML DBMS would be good for storing hierarchical data, but still, quite some interfacing and conversion code is needed to connect to anything based on OO.

A tree based store has the potential of being the fastest when enough time is available for crafting custom code for each "query". Fast, but not easily extensible.

The RDBMS solution is the current, popular, solid, proven but mediocre solution. It's overall performance is expected to be reasonably fast. But there's still some custom code to be written for interfacing with the OO world.

The Object/relational solution is perceived as a temporary solution. Until OODBMS has matured, it's an easy solution for storing objects. Unfortunately, the price paid for this automagical conversion layer is speed.

Remaining is the OODBMS which also doesn't need any custom code. It looks as if OODBMS is still more of a promise than a usable DBMS, as there are still few implementations. There are plenty of reasons why OODBMS is not catching up, but nobody seems to agree on anything. Some say it's because it's more important to make the data portable than it is to make the code portable, and an OODB is too tightly coupled to the OO Language implementation. Ten years from now, no one will care what the code looks like, but care tremendously what the data looks like. Others say it's just not worth the time learning yet another DBMS where the RDBMS works good enough.

3.4 Caching

This section introduces a definition of a cache (section 3.4.1), the layers of cache in a computer system (section 3.4.2) and its relation to database benchmarking (section 3.4.3). This information helps interpreting the database measurement results in chapters 6 and 7.

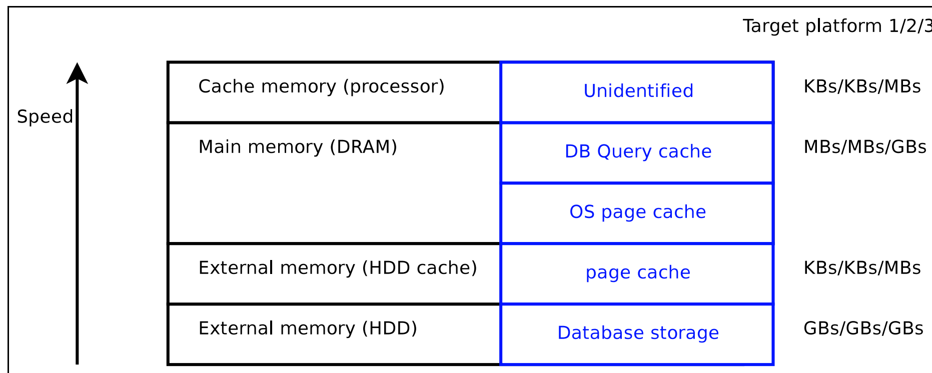

Figure 3.5: Layers of storage and cache, black boxes represent physical hardware stores, blue boxes represent usage of those stores determined by software.

3.4.1 Definition of a cache

A cache[20] is a duplicate (redundant) collection of data from which the original values are stored elsewhere. Using a cache makes sense where the original data is expensive (usually in terms of access time) to fetch or compute relative to reading the cache. Once the data is stored in the cache, future operations can be performed on the cached copy rather than refetching or recomputing the original data.

3.4.2 Layers of cache

In a personal computer, there a several places in which data can be stored. The processing unit is connected to all these stores. The primary and secondary stores and their relative speed is visualized in the first column of figure 3.5. As a rule of thumb, a faster store is more expensive and smaller (in both capacity and physical size) than a slower store.

Software controls what data goes in which store. Software may designate (a part of) a fast store as a cache for data that is kept in a slower store. Control by the operating system, but also in user applications, happens all the time. Take a smart DBMS for example, it knows (the limits of) the environment, adapts and takes advantage. Some of its algorithms are optimized for RAM, others are optimized for disk-based storage. An efficient approach requires a different strategy every time.

3.4.3 Database benchmarking and cache

The ways in which the OS, the DBMS and other applications interact and handle their cache can be very complicated. For the end-user performing tasks on a personal computer, caching usually enhances performance. But for performing accurate, repeatable benchmarks it's a major hurdle. In an effort to separate the "first-time" performance costs from the cached ones, all caches have been identified, as visualized in the right column in figure 3.5. The next step is to control the caches. Unfortunately, some caches are designed to be transparent, so they're difficult to control (and clear out).

Ideally, before performing a measurement, all caches should be flushed to ensure there are no traces of any previous measurements. Ensuring such a clean-room environment seemed impossible. Although most pieces of the puzzle are now visible and theoretically controllable, the thesis was long overdue. Therefore, all knowledge on controlling caches is collected for future reference in section 9.3.4.

[20]More information about caching: `http://en.wikipedia.org/wiki/Cache`

Chapter 4

Benchmarking and framework design

4.1 Introduction

One way to get a general impression which popular DBMS works best, is by reading its reviews and commercials. But the only way to really find out which DBMS works best in a certain situation is to hunt for both popular and unpopular (to include those without a marketing strategy) DBMSs and just try them out. The process of "trying them out" can be formalized as benchmarking, which should be looked upon as a tool for testing and comparing DBMSs.

Starting with some theory in the following sections, the definition of benchmarking in general and in the field of computers is explained in sections 4.2 and 4.3. In section 4.4, the spotlight is on current benchmarking standards. Finally, the ways in which a benchmark can be conducted are explored in section 4.5.

From section 4.6 on, the design for the database benchmarking framework is discussed.

Finally, in chapter 5, the framework design described in this chapter is implemented.

4.2 The benchmark definition

By the original definition[1], a benchmark is a point of reference for a measurement. The term originates from the chiseled horizontal marks that surveyors made into a wall from which an angle-iron could be placed to bracket (bench) a levelling rod, thus ensuring that the levelling rod can be repositioned in the exact same place in the future.

Presented below is a list of benchmarking benefits[2] that will lead to quality improvement:

- Exposes need for change
- Framework for change
- Demystifies change
- Decreases subjectivity
- Legitimizes targets

[1] `http://en.wikipedia.org/wiki/Benchmark`
[2] More on `www.ecu.edu.au/Quality@ECU/ecuonly/tools/benchmarking.doc`

- Generates new ideas

In businesses[3], benchmarking helps to improve process effectiveness, product quality and service delivery. It enables an organization to compare their existing performance and approach to others, and identify elements that can be adopted and adapted in their business context. Benchmarking enables organizations to compare and improve themselves and prompt innovation. Although benchmarking is widely used, it not a precise science and there are many different approaches.

In computing, a benchmark is the result of running a computer program, or a set of programs, in order to assess the relative performance of an object, by running a number of standard tests and trials against it. The term, benchmark, is also commonly used for specially-designed benchmarking programs themselves. Benchmarking is usually associated with assessing performance characteristics of computer hardware, for example, the floating point operation performance of a CPU, but there are circumstances when the technique is also applicable to software. Software benchmarks are, for example, run against compilers or DBMSs.

Benchmarks are designed to mimic a particular type of workload on a component. ***Synthetic*** benchmarks do this by specially-created programs that impose the workload on the component. ***Application*** benchmarks, instead, run actual real-world programs on the system. Whilst application benchmarks usually give a much better measure of real-world performance on a given system, synthetic benchmarks still have their use for testing out individual components.

Benchmarks can be deceptive if they're not described and performed with extreme care and precision. Also presenting and interpreting the results without extreme care will easily lead to misinterpreted conclusions. On purpose or not, uncareful interpretation of a benchmark and its results may lead to bench-marketing.

4.3 Benchmarking properties

To sum up the generally desirable attributes that apply to benchmarking, they're listed below[4]:

- Relevance (meaningful within the target domain)
- Understandable
- Good metric(s)
- Scalable (applicable to a broad spectrum of hardware)
- Enough coverage (no oversimplification of the typical environment)
- Acceptance (all stakeholders embrace it)
- Repeatable (results are solid)

The following precautions apply to benchmarking in general:

- No single metric possible (domain specific)
- The more general the benchmark, the less useful it is for anything in particular
- A benchmark is a distillation of the essential attributes of a workload
- Benchmarks can become counter productive by encouraging artificial optimizations
- Even good benchmarks become obsolete over time

[3] `http://www.ogc.gov.uk/sdtoolkit/reference/documentation/p24_bench.html`
[4] A comprehensive overview at `http://www.tpc.org/information/sessions/sigmod/sld003.htm`

4.4 Standards in benchmarking

The Standards Performance Evaluation Corporation[5] (SPEC) is a non-profit organization that aims to produce fair, impartial and meaningful benchmarks for computers. SPEC was founded in 1988 and is financed by its member organizations which include all leading computer and software manufacturers. SPEC benchmarks are widely used today in evaluating the performance of computer systems; the results are published on the SPEC web site[6].

Unfortunately, SPEC doesn't cover any DBMS benchmarks. This void is filled by the highly regarded Transaction Processing Performance Council (TPC). The TPC provides DBMS testing suites that compare a DBMS coupled to specific hardware. The pairs are compared in terms of performance and price/performance ratio. The TPC-C[7] benchmark simulates a complete computing environment where a population of users executes transactions against a database. The benchmark is centered around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. TPC-C is clearly an application benchmark, it simulates real-world business applications. It's the complex result of more than 2 years of development. Its specification defines a balanced, representative, database and IO intensive OLTP[8] mix.

Another performance measurement organization is the EDN Embedded Microprocessor Benchmark Consortium (EEMBC[9]). The EEMBC is dedicated to "a collaborative effort in developing a suite of performance benchmarks that will target key applications of embedded systems", and provide an objective means for evaluating processors and controllers. Typical available benchmarks are: FFT, JPEG (de)compression, MPEG encoding and decoding, cryptography. These are all examples of synthetic benchmarks. It seems that a DBMS is not a key application of an embedded system as there are no DBMS related benchmarks provided by the EEMBC.

Unfortunately, but naturally, the self-appointed benchmark authorities don't provide free tools to carry out or even reproduce any benchmarks[10]. Nonetheless, they provide a good source of information.

4.5 The benchmark procedure

The steps to conduct a successful benchmark are shown in figure 4.1.

Benchmarks can be performed in several ways. The pros and cons of each option are discussed below.

Manual or automated benchmarking:

- Manual benchmarks require expertise for every run, automated benchmarks mainly during design.
- Manual benchmarks are faster to setup, but each run requires work.
- If the benchmark is to be run often, automated benchmarking saves the setup time.

[5] `http://en.wikipedia.org/wiki/Standards_Performance_Evaluation_Corporation`

[6] `http://www.spec.org`

[7] `http://www.tpc.org/information/benchmarks.asp`

[8] On Line Transaction Processing. Applications include electronic banking and order processing. Complete definition at: `http://en.wikipedia.org/wiki/OLTP`

[9] `http://www.eembc.org`

[10] The Standard Oracle License Agreement (`http://www.oracle.com/technology/software/products/database/oracle10g/htdocs/linuxsoft.html`) states: "You may not disclose results of any program benchmark tests without our prior consent;". A quick scan on the Internet revealed that a few benchmark sites had Oracles results removed after being contacted by Oracle because of the license. One benchmark including Oracle on rank 2 was found, from the TPC itself! This smells fishy.

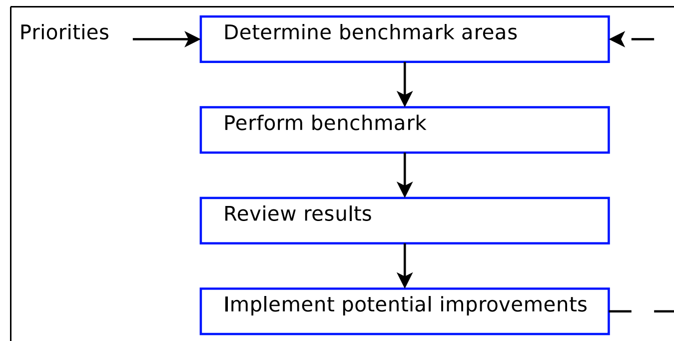

Figure 4.1: The benchmark procedure

- Reproducing[11] results is easier using automated benchmarks.
- Automated benchmarks may be less error prone, if set up correctly.

Several specific benchmarks or one generic benchmarking framework:

- A generic benchmarking framework is expensive to create, but after that, appending a new test to the framework is cheap.
- Writing one specific benchmark is cheap, compared to adding a new test to the framework.
- writing several specific benchmarks is expensive, compared to adding several new tests to the framework.
- A framework enables structured reuse of work.

For the sole purpose of writing the thesis, I would've chosen to build an automated benchmark without the framework. This allows me to minimize the time spent on benchmarking. However, high on the priority list of Philips is creating a solid base for doing many benchmarks, now and in the future. This means an automated benchmarking framework that enables adding a new test easily without much expertise. It's the most expensive route, in short term. But Philips benefits in the long run.

For these reasons, an automated benchmarking framework will be designed and implemented later on.

4.6 Defining the benchmarking area

To start the benchmarking process, the benchmarking area has to be defined. By taking the goals and priorities from chapter 2 and the time-frame of appendix F into account, this area can be defined. Leaving behind the requirements that can't be benchmarked, the following list defines the ideal benchmarking area:

- Performance
- Scalability

[11]Exact reproduction is an utopia. It's unavoidable to get different results even by altering what appears to be very minor factors. This is especially true in embedded benchmarking, where variables such as board bus speed, clock speed, memory speed, memory set-up, memory wait-states, types of memory, compiler used, compiler flags used, incorrect programming of the on-chip timer, and literally dozens of other factors can make a huge difference in performance (from EEMBC).

- Platform independence
- Memory footprint
- Energy usage

This area of aspects to benchmark is too wide, scrubbing is required. Energy measurements are beyond the scope of a benchmarking tool which exists only in software. Memory measurements are difficult to perform accurately. These areas are deferred to the list of future work in section 9.

Now that has been defined *what* to benchmark, the benchmarking area can be determined by defining *how* to benchmark.

Performance will be measured by the time it takes for a testcase to complete. While this number is very much dependent on the hardware it runs on, all DBMSs will run under the same circumstances which makes them comparable.

Scalability will be measured by running a variable set of testcases with a variety of dataset sizes. This is benchmarked on different hardware to see the effects of a limited amount of memory and processor speed. Also the database schema can be switched to simulate simple or complex scenarios.

Platform independence will be proved by running the benchmarks on different hardware architectures as demonstrated in chapter 7.

To provide all this flexibility, the framework has to be modular. Every module should allow a third party to create an interchangeable module to ensure flexibility and extensibility in the future.

These are the areas that should be modularized:

- DBMS abstraction - allow benchmarking of multiple databases
- Testcase abstraction - allow different testcases to be measured
- Dataset abstraction - generating datasets of a certain size on demand
- Schema abstraction - allow different schemas to be benchmarked

4.7 The actual benchmark

In the second step: performing the benchmark, an instantiation of the use-cases that were defined in section 2.8 is benchmarked. Each use-case will become a separate testcase in the benchmarking framework, which will be implemented in SQL for the RDBMSs and custom code for other DBMSs.

The first series of tests (performed in chapter 6) represent a set of general testcases that are formed in such a way that they'll work on the lowest common denominator. They'll be understood by every DBMS benchmarked by the framework and will work on virtually any new DBMS that will be appended in the future.

The second series of tests (performed in chapter 7) includes optimizations for each individual DBMS.

4.8 Desired benchmark results

As the third step recommends, the results have to be reviewed. If one is satisfied with the results, the benchmark is completed, otherwise repeat from step one.

This section defines what the desired results of the benchmarking framework should be:

- An indication if the benchmark was successful and accurate enough.
- Data indicating the performance (time-wise) of a benchmark.
- Data indicating the scalability of each DBMS.
- How well the DBMSs compare to each other.

Chapter 5

Benchmarking framework implementation

5.1 Introduction

Chapter 4 specified the benchmarking framework. This chapter describes the implementation of the benchmarking framework.

The benchmarking framework is ideally an application that given several parameters, benchmarks one DBMS and reports how well the DBMS performed (see figure 5.1). The design in chapter 4 talked about conducting series of tests. This is accomplished later on by using the components implemented in this chapter in a mass benchmarking script. To allow maximum flexibility, this functionality is kept outside the framework.

The core of the benchmarking framework is implemented in the C++ language. The Storage and Applications group is traditionally C++ oriented due to its embedded software expertise. Although I preferred acting on the "best tool for the job" mantra, there is value in a "standard" solution like C++ that anyone in the group can work on. Using a higher level programming language such as Java or Python[1], in which I had more experience, would bring several benefits:

- More powerful. The standard libraries have much more tools than the standard C++ libraries. All of this is portable unlike some 3rd party C++ libraries that would otherwise be required to offer a similar level of functionality.

[1] An interpreted, OO programming language: `http://www.python.org`

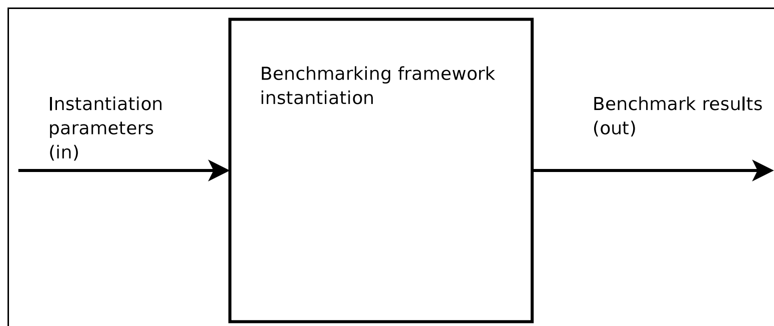

Figure 5.1: Benchmarking framework diagram.

- Easy writing. Ease of writing reduces faulty code. Also garbage collection saves writing considerable lines of code.
- Better maintainable. The code is very readable and easier to redesign and modify.

Both are interpreted languages, but most python core functionality is written in C++ with a small interpreted layer on top. Speed is therefore slower than a functionally identical C++ application. But speed is not an issue in this benchmarking framework. No time-consuming operations are performed by the benchmarking framework. It's mostly telling the DBMSs what to do.

Another language can't outweigh the benefits of using C++ as primary language for the benchmarking framework development. The top reason being maintenance of the benchmarking framework by future programmers that has to be guaranteed. Therefore C++ is chosen as the primary programming language.

5.2 Architecture

The benchmarking framework as shown in figure 5.2 is divided into several components as was designed in chapter 4:

- *Overseer*: is a script, written in BASH[2], that reads the parameters for the benchmark instantiation from a configuration file. This script instructs the other components when to start, in which order and with what parameters.
- *DatasetGenerator*: is a script, written in Python, that generates a dataset of a certain size to be used by the BenchmarkControl component. Its inner workings are explained in section 6.2.
- *BenchmarkControl*: is the most critical component of the framework, written in C++. Communication with the DBMS under test is handled by this component. Under the conditions specified by the parameters, BenchmarkControl operates and benchmarks the DBMS under test by connecting to the DBMS through its API and timing the performed commands.
- *Ploticus*[3]: takes the raw data generated by BenchmarkControl and visualizes it in a series of graphs that will present the results of a successful benchmark.

[2]A command language interpreter: `http://www.gnu.org/software/bash/bash.html`
[3]`http://ploticus.sourceforge.net`.

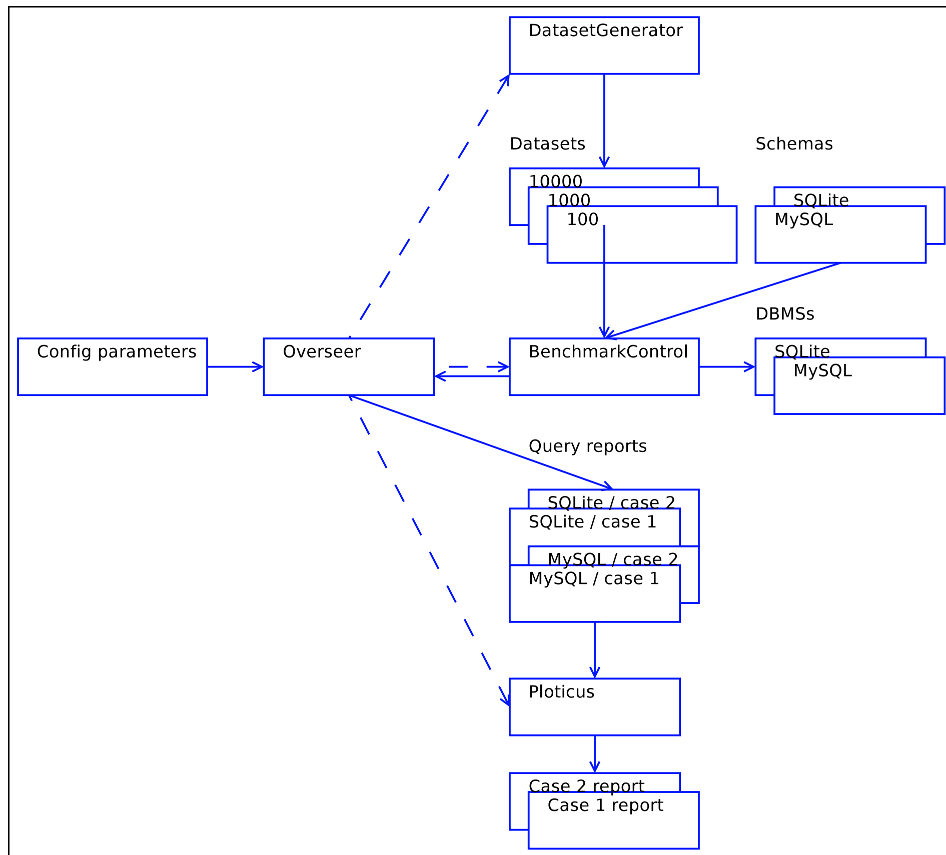

Figure 5.2: Benchmarking framework architecture. The solid lines indicate data flow. The dotted lines indicate a control channel.

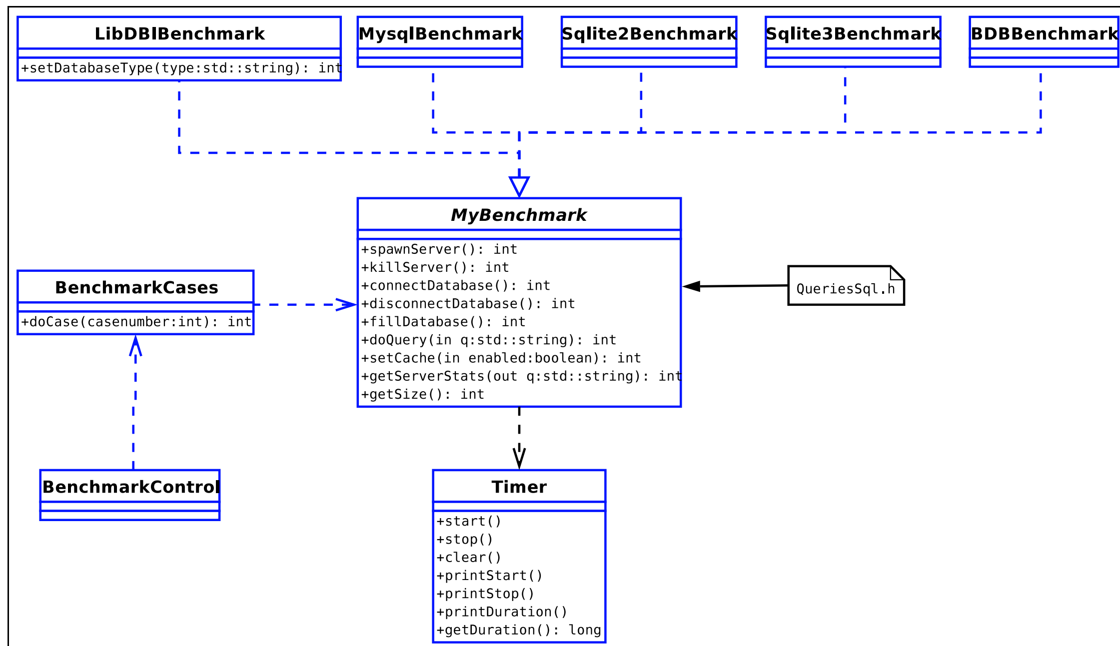

Figure 5.3: Class diagram of the BenchmarkControl component in figure 5.2.

Figure 5.3 shows the BenchmarkControl component zoomed in. It is made out of the following classes:

- *BenchmarkControl*: parses the command line arguments and starts the testcase for the chosen DBMS.
- *BenchmarkCases*: breaks down the testcase into the several lower-level operations to be executed sequentially in an instantiation of MyBenchmark.
- *MyBenchmark*: is an abstract class that defines the various operations[4] that should be executable on each implemented DBMS. By using abstraction, each (future) DBMS can be handled in an uniform way.
- *MysqlBenchmark*: is one of the implementations of the MyBenchmark class (implemented functions not shown in the class diagram.) It implements all operations defined in MyBenchmark so that every current and future testcase is able to run on the MySQL DBMS.
- *Timer*: is a helper class to conduct timings of DBMS operations.

Classes like LibDBIBenchmark, MysqlBenchmark, Sqlite2Benchmark and Sqlite3Benchmark are simple wrapper classes, for most operations. They map almost directly to the underlying DBMS APIs shown in figure B.3. BDB is the exception in which the BDB API is lacking rich functionality compared to the other DBMS APIs. To compensate, source code had to be written to reach a sufficient level of functionality.

5.3 Features

Great efforts have been made to create a benchmarking framework that is robust, powerful, fully automated and easy to use.

[4] At the time of writing: start, stop, connect, disconnect a DBMS. Fill and query a database.

Robustness

The source code is fully documented, easy to understand and well tested. Whenever a testcase fails, the framework notices this but continues to operate and finishes the other tests anyway. After completion of the whole run, a summary of errors is presented to the framework operator. The operator then knows how to interpret the results or discard them completely.

By default, measurements are performed multiple times to guarantee accurate results. If the framework is unable to deliver accurate results, it'll report how (in)accurate the measurements were. This works as follows: the standard deviation of a set of measurements should not exceed a certain threshold. The framework reruns the measurements multiple times again until the standard deviation is small enough. If this won't happen within a certain amount of tries, the out of the ordinary standard deviation value is the result of DBMS behavior, not any intermittent distortion. Therefore the threshold will be lowered in the next rerun until the smallest possible standard deviation is accepted as a valid measurement[5].

Powerful

Many parameters can be specified as input for a benchmark run:

- Which testcases to run. One or more (values: 1..7)
- Which DBMSs to test. One or more from (values: BDB, MySQL, Libdbi_MySQL, SQLite2, SQLite3, Libdbi_SQLite)
- Which datasets to use. One or more from (values: 10, 20, 40, 100, 200, 400, 1000, 2000, 4000, 10000, 20000, 40000, 100000)
- Which schema to use. One schema definition file for each DBMS.
- Moment of measurements. The first time a query runs (cold) or the second time (warm) to benefit from caching.
- How many iterations of the same measurement should be performed for one point in the graph (with mean and standard deviation if > 1). ($\mathbb{N}$)
- Enforce a limit on the amount of returned results from a testcase. Unlimited or ($1..\mathbb{N}$)
- Weights for each testcase to generate an aggregated graph combining all testcase results.

These are the generated results after a successful benchmark run:

- A graph per testcase. The time to complete a testcase on the y-axis. The dataset size on the x-axis. The graph is formatted in threefold. The first one with linear axes, the second with a linear x-axis and logarithmic y-axis and the third one with both axes on a logarithmic scale. Depending on the situation, one may more accurately visualize a phenomena than another format.
- An aggregated graph combining all testcase results. Each testcase has been assigned a weight to represent its importance. The total time to complete all testcases (the time for each testcase is multiplied by the testcase weight factor) on the y-axis. The dataset size on the x-axis.

[5]In the later versions of the framework that run in a clean test environment, this mechanism is barely invoked anymore

5.4 Extensibility

The framework is easy to extend in almost every way.

- Adding a new testcase for RDBMSs is as easy as adding a new SQL query to the testcase file. For DBMSs such as BDB it requires programming work as it doesn't have an SQL interface.
- Adding a new DBMS to test requires a new derivative from the MyBenchmark class. If the DBMS is an RDBMS and has a complete API, it will take no longer than one day of work. Addition of a special DBMS without a featureful API may require more work.
- Adding a new dataset size is trivial.
- Changing the dataset generator to generate other data is a simple task, knowing that the current dataset generator is only a script containing 20 lines of source code.
- The difficulty of letting the framework make new analysis's or graphs is dependent on the intended type and complexity. For example, generating a graph for each combination of DBMS and testcase with time on the y-axis and dataset size on the x-axis requires only a few lines of source code additions to a script (no need for recompiling the framework):
  - Filtering the needed timings from all measurements (2 lines of source code).
  - Parsing or normalizing the measurements (not needed in this example).
  - Merging all filtered results into a table of data (2 lines of source code).
  - Visualizing this table as a graph (1 line of code).

5.5 Debugging

Writing status information to the terminal during benchmarks has impact. The extra time it takes to write a line depends on the terminal. Pure text based terminals are fast, but graphical terminals like xterm or gnome-terminal are very slow. Anyway, it's necessary to have output sometimes, for example to check if the query results are correct. That's why the benchmark framework can be run in the so called DEBUG mode. It's a compile-time option to make sure that if the DEBUG mode is disabled, there's absolutely no unnecessary code produced that could influence the benchmark.

When DEBUG mode is enabled, information about problems, errors or other important messages will always appear. Needless to say, any benchmark performed in DEBUG mode should be regarded as unreliable. Messages displayed on the terminal have the following prefixes (if any):

- "ERROR:" to STDERR, this message will appear with and without DEBUG mode. Measurements are unreliable.
- "WARNING:" to STDOUT, this message will appear before or after measurements, with and without DEBUG mode. Measurements are initially unreliable.
- No prefix to STDOUT, this message will appear before or after measurements, with and without DEBUG mode. Measurements are reliable.

5.6 Benchmarks

Three series of benchmarks were conducted using the framework.

- The first is an initial experiment (documented in appendix B) that consists of 3 DBMSs and 3 queries. The purpose of this run is to gain confidence in the framework, as a proof of concept. The framework is improved during and after running the benchmark so it'll be ready for better and larger tests. The reason for this first benchmark run being an appendix is that the measurements are very unreliable, and therefore contain invalid data. The reason for not discarding this chapter altogether is for future developers on the framework. It identifies several problems encountered while benchmarking.
- The first accurate benchmark run (chapter 6) benchmarks 2 more DBMSs and 4 more queries than the initial experiment. The benchmarking framework allows larger datasets, different schemas, is more efficient and reliable overall. Useful conclusions can be drawn from the measurements at this stage. The purpose of this run is comparing the out-of-box DBMS performance.
- The second benchmark run (chapter 7) is a playground for DBMS optimizations and unique DBMS features. Now the DBMSs are really pushed to the limit. While interpreting these results, it has to be taken into account that the DBMSs may be no longer directly comparable because of different test conditions and environments. The purpose of this run is optimizing the DBMS and see if tuning substantially improves performance compared to the out-of-box performance.

Chapter 6

Benchmark result analysis 1

6.1 Introduction

In this chapter, all use-cases mentioned in section 2.8 are translated to the query language of the DBMSs that are tested. The expected results of a testcase along with the measured results are discussed for every testcase.

At the end of the chapter, all testcases are aggregated into a new graph where the DBMSs are ordered by the overall performance rating.

In this chapter, the DBMSs are benchmarked with their default out-of-the-box settings. If a DBMS didn't perform as expected, it is tuned in chapter 7 to see if the results can be improved.

This is the first accurate benchmark run after the initial experiments in appendix B. During the initial experiments, most problems have been identified and repaired. Also some improvements have been made in several areas:

- SQLite version 2 is added to the framework
- The first non-SQL database, Berkeley DB (BDB[1]) is added to the framework
- The framework is more accurate in its timing measurements due to stripping more overhead[2] from the timings
- The framework has become more generic through abstraction of the schema (default schema is now an HDD60-subset schema)
- The framework has become more generic by providing a user defined limit on the amount of returned values
- A new dataset generator is used, simulating better real world data
- The framework is able to indicate inaccurate measurements using standard deviation over several measurement repetitions and a (dynamic) time threshold as a last defense against measurements influenced by interference.

[1] BDB is explained in section 3.3.2

[2] Sometimes a try..except construction was used to handle certain situations. Not only for entering the construction on errors, but also during a measurement. Like writing text to a graphical terminal, this operation also contributes to inaccuracies. After removing these constructions, a speedup of at least 1.8ms per construction was reached.

6.2 Benchmark parameters

Hardware platform:

At the time of performing this benchmark, the framework was not yet working[3] on an embedded platform, so only target 3 (home server) is used as hardware platform.

DBMSs:

This run, the following DBMSs are benchmarked:

- Berkeley DB
- MySQL
- MySQL with Libdbi
- SQLite v2
- SQLite v3
- SQLite v2 with Libdbi

For comparing SQLite 2 to SQLite 3 with Libdbi for the same versions of SQLite, SQLite v2 was added[4].

MDB is excluded at this time[5]. It's analyzed separately in appendix C. The reasons for excluding MDB are:

- Slowness. It effectively limits the dataset sizes to be tested.
- Decreased number of framework dependencies. The MDB framework requires several external libraries.

Only without MDB, the framework can be compiled and used on embedded devices, as will be demonstrated in the next benchmark run in chapter 7. Cross-compiling the framework for a different architecture than IA32 (the PC architecture) is substantially easier. A partner of Philips Research, CWI[6] is also planning[7] to use the framework and add more databases to it. Third parties can't be expected to compile MDB without trouble since it drags in a lot of other dependencies. In addition, there are also intellectual property issues.

To wrap up this issue: to keep the benchmark framework more accessible, maintainable and simpler, MDB is taken out.

Schema:

The schema remains the same one as used during the initial experiments (see figure D.1).

Datasets:

The dataset sizes used in the benchmark are: 10, 20, 40, 100, 200, 400, 1000, 2000, 4000, 10000, 20000 and 40000. The new datasets being: 10000, 20000 and 40000 assets. The contents of these datasets differ from the initial experiments. Now that the file importer can't be used anymore due to MDB removal, a new dataset generator has been developed.

[3] In chapter 7, an embedded platform is benchmarked.

[4] Because there was no driver for SQLite 3 in Libdbi when work on this thesis ended.

[5] Some time after this benchmark run was finished and analyzed, my skillful colleagues helped bringing back MDB into the framework in a way that it is now easily enabled and disabled using a compile-time option.

[6] Centrum voor Wiskunde en Informatica, `http:\\www.cwi.nl`

[7] Just before ending this thesis, the preliminary XML DBMS results were received and could be included in chapter 7.

The new dataset generator has the following advantages over the old file importer that was described in appendix B.3.1:

- More balanced dataset generation. Assets to album and artist ratio is a parameter (default 10:1). This resembles real-world albums more closely. They often contain around 10 songs.
- Faster. Large datasets are created almost as fast as the harddisk can write.
- More practical. To (re)create a dataset, no MP3 files are needed anymore. The intermediate format is simply created from a simple set of rules[8].
- The source code is easier to understand.

There are also disadvantages:

- The generated dataset is synthetic. Not all albums only have one artist and a constant number of tracks in reality.
- The python dependency is appended to the framework. Although it's very portable[9].

Dimensions:

As described in section 5.3, several graphs are produced for every test. The y-axis always represents the elapsed time from the moment of executing a query until receiving the last result. The x-axis always represents the dataset size that the database is filled with. The graphs contain logarithmic scales so the measured datasets will be evenly spaced.

Iterations:

The benchmark is performed three times, for every testcase. Each time under a different circumstance:

- Cold run, unlimited results, 1 iteration[10].
- Warm run, unlimited results, 5 iterations.
- Warm run, amount of results limited at 100 records, 5 iterations.

For more information on cold and warm runs, see the framework features in section 5.3.

6.3 Testcase 1: database creation and filling

Purpose of this testcase:

Measuring the performance of the first use-case: Joe bought a brand new PIC. He wants to copy all his music from his laptop to his PIC.

The expected outcome of this testcase:

Because simple data transportation from the caller application to the database is the bulk of the query, linear behavior is expected.

[8] Asset to album and artist ratio, amount of assets to create, dummy asset and album names.

[9] It's running on the embedded platform already, for the third run. Portability notes: `http://www.python.org/doc/faq/general.html`

[10] The framework needs modification for allowing multiple iterations during a cold run. This is because the test failure indication mechanism isn't able to rerun a test in case of suspected failure during a cold run. This issue is thoroughly documented within the source-code.

The query in words: insert all assets (and related data) into the database.
The query in SQL: INSERT INTO Artists VALUES (...); and INSERT INTO Albums VALUES (...); and INSERT INTO Assets VALUES (...);

The results are visualized in figure 6.1. Note that the two graphs are similar and only differ in used scale. This serves as an example to get used to interpreting graphs using the logarithmic scale.

Observations and conclusions drawn from the results of this testcase:

BDB and MySQL both perform and scale very well.

All SQLite variants have serious problems when the amount of inserted data grows beyond the smaller datasets. Either SQLite is slow at just storing the data or updating the index (or both). To find out the exact cause, SQLite and MySQL are benchmarked again but this time without creating an index on every insert. The results are shown in figure 6.2. Creating indexes causes a very small performance hit. The effect of having no indexes available for a SELECT query is shown in testcase 3, section 6.5. In this testcase, having an index on *tracknr* is a great advantage. Therefore testcase 3 makes a good comparison.

The reason for the slow SQLite behavior according to their developers is that because SQLite doesn't have a central server to coordinate access, SQLite must close and reopen the database file, and thus invalidate its cache, for each transaction. In this test, each SQL statement (n in total, where n is the amount of assets) is a separate transaction so the database file must be opened and closed and the cache must be flushed n times. SQLite calls fsync() after each synchronous transaction to make sure that all data is safely on the disk surface before continuing.

Remember when during the initial experiments in appendix B, MySQL had an unique feature where all data could be inserted at once by one big INSERT statement. SQLite wasn't able to do this. There is a way "around" this that speeds up SQLite: by using one big transaction so that SQLite does not have to do any fsync()s until the very end. This works with the following SQL statements:

BEGIN;
INSERT ...;
INSERT ...;
...
COMMIT;

The results of using a transaction are visualized in figure 6.3. This test is performed once with and once without index creation. As can be seen in the graph, this makes SQLite the fastest DBMS at bigger datasets. Also the performance of index creation benefits from transactions as the index can now be created as one operation at the end of the transaction.

Note that BDB still isn't tuned at this time.

6.4 Testcase 2: simple scan

Purpose of this testcase:

Measuring the performance of the second use-case: John wants a quick overview of all music on his PIC. He just wants a list of songs.

The expected outcome of this testcase:

Because simple data transportation from the database to the caller application represents the bulk of the query, linear behavior is expected.

The query in words: return the complete details of all assets. (Indicates raw performance.)
The query in SQL: SELECT * FROM Assets

The results are visualized in figure 6.4.

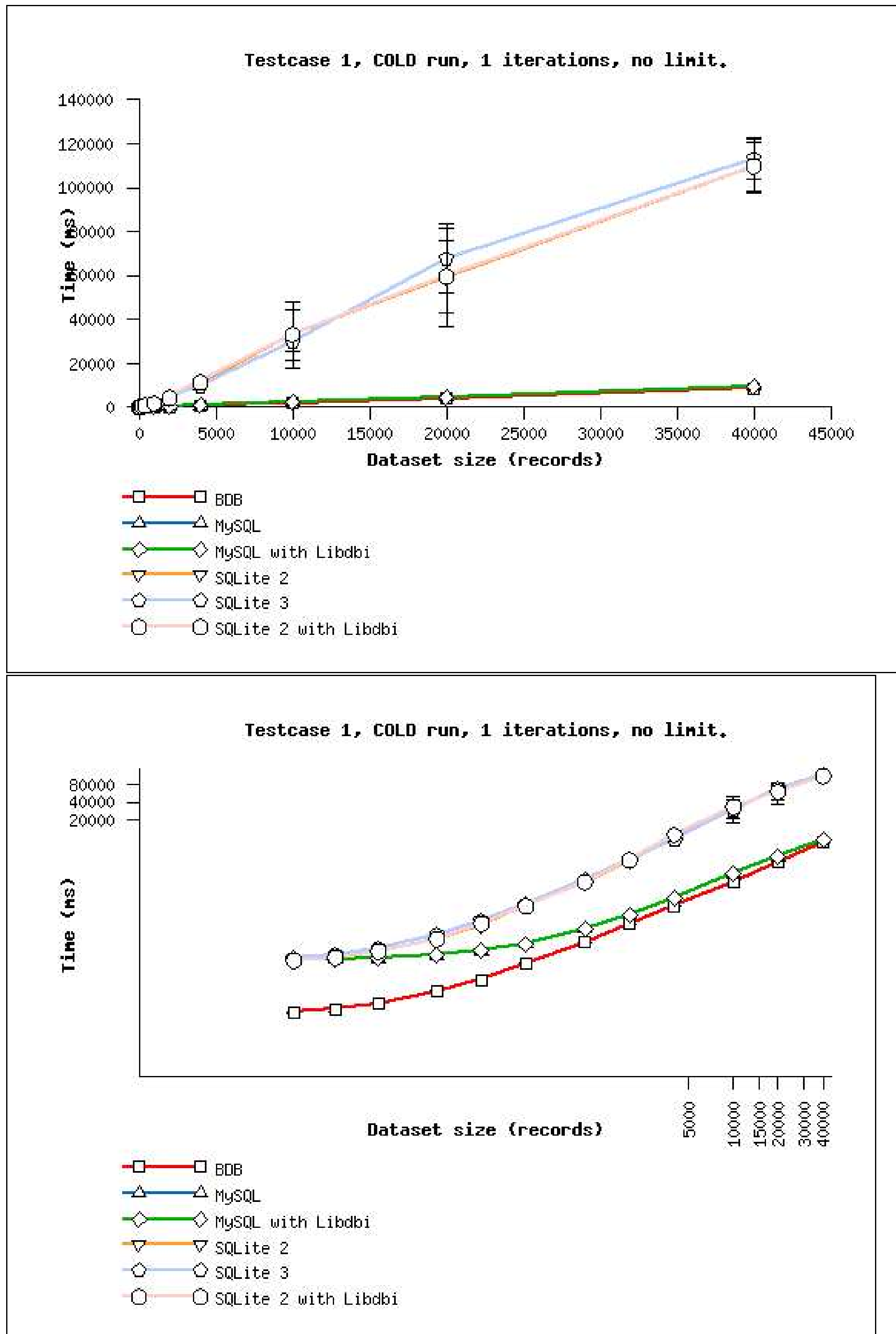

Figure 6.1: Analysis 1, case 1. Total measured INSERT time per dataset in a fresh database.

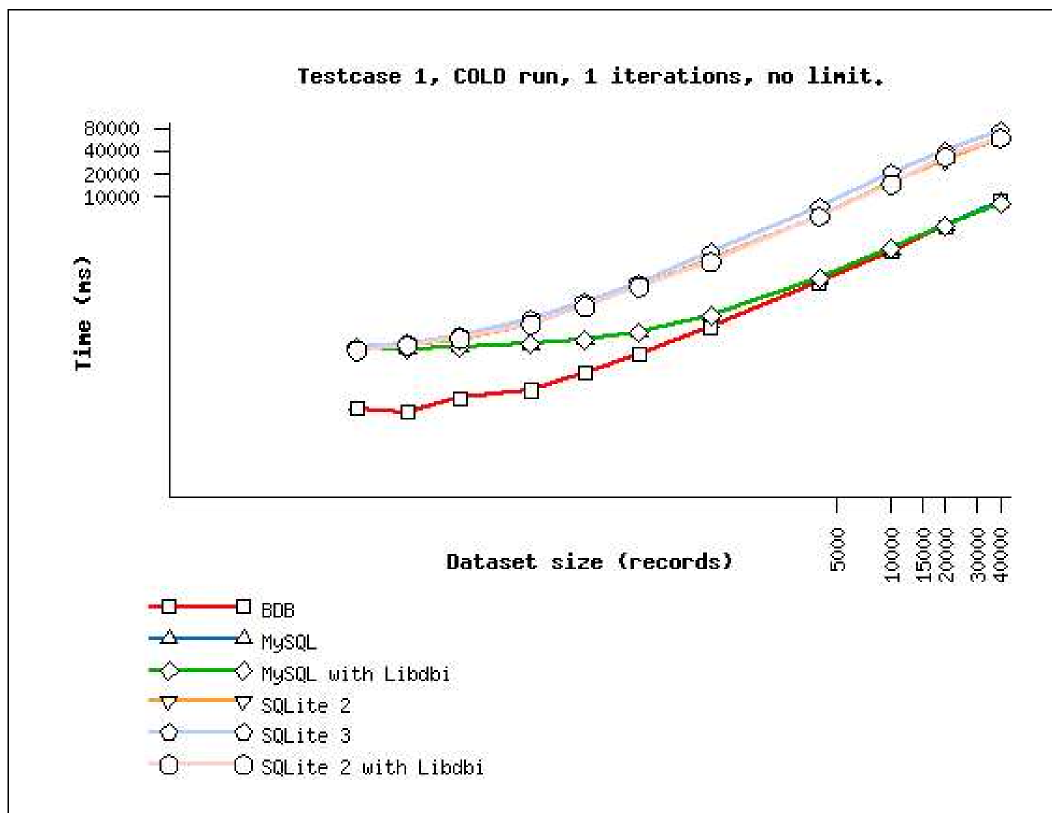

Figure 6.2: Analysis 1, case 1. Total measured INSERT time per dataset in a fresh database without creating indexes.

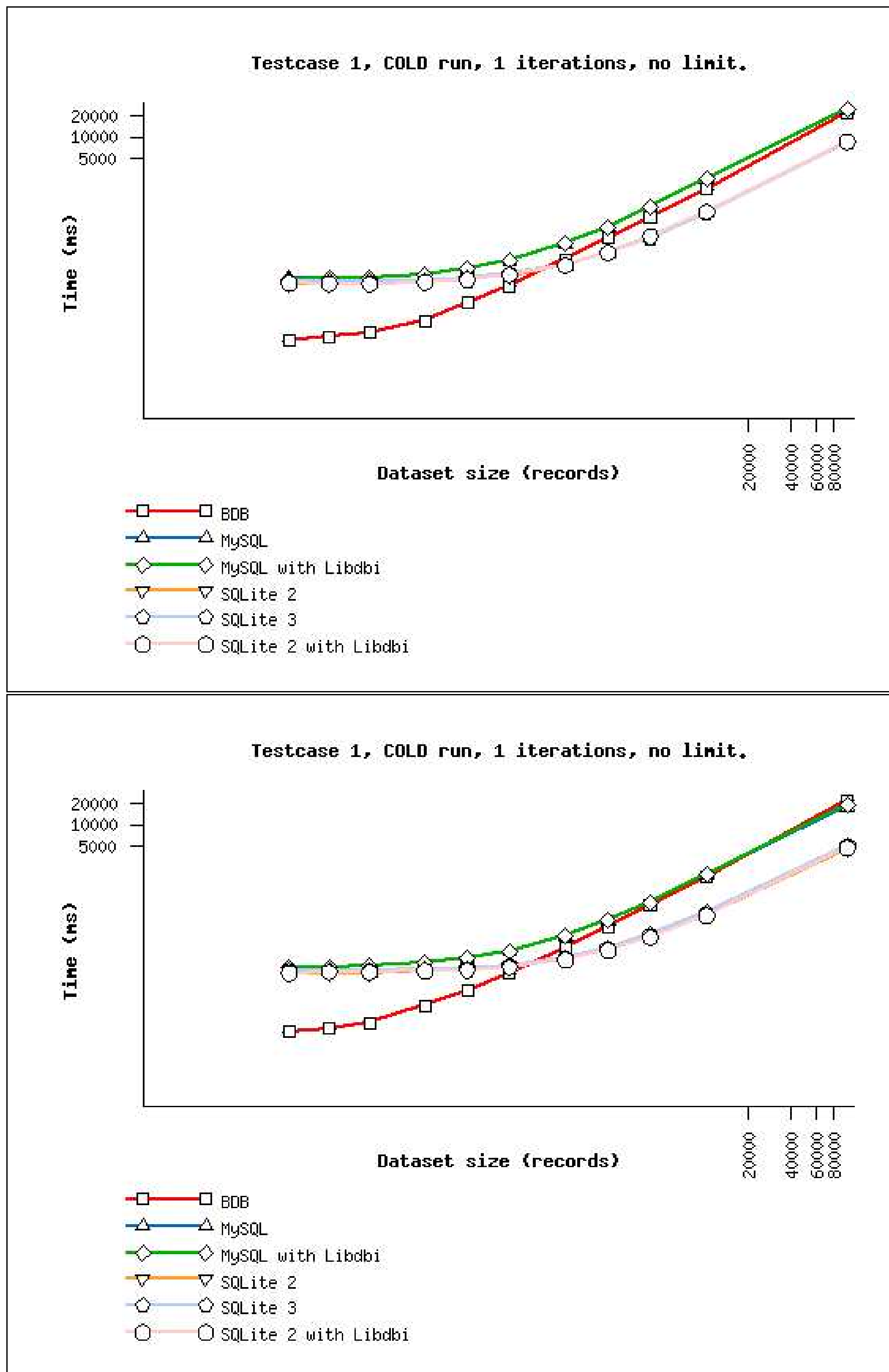

Figure 6.3: Analysis 1, case 1. Total measured INSERT time per dataset in a fresh database using a transaction, with (left graph) and without (right graph) creating indexes.

Observations and conclusions drawn from the results of this testcase:

During both the unlimited cold and warm runs, BDB performs best. Closely followed by SQLite 2 and the other SQLite variants. MySQL takes at least 4 times more time than BDB in every instance. What's noticed during the warm run is that MySQL makes good use of caching which results in a substantial speedup using small datasets.

The effect of caching can be modeled as: $ax + b$, where a represents the time spent by retrieving records from a store, b represents the initial DBMS startup cost and x represents the dataset size. The a component will be much higher during the cold run as the data has to be retrieved from the harddisk. During a warm run, the a component will be lower as (part of) the results will be retrieved from memory. Same holds for the b component.

This measurement confirms the claims stated on the SQLite website. SQLite 3 is still slower than SQLite 2. This probably won't change much in the future as SQLite 2 is just simpler in design and less featureful.

As expected, the Libdbi layer on top of SQLite 2 accounts for some overhead. This is not the case for Libdbi with MySQL, which demonstrates a better integration.

When the amount of results is limited at 100, all DBMSs perform really well. While the measurement results look a bit unstable, they're all below 1ms. That seems to be the limit to which the benchmark environment is not influencing the measurements too much. Further environment stabilization requires expert Linux OS knowledge.

6.5 Testcase 3: simple scan with filter

Purpose of this testcase:

Measuring the performance of a successful third use-case: Janet wants to search directly for a song title and play it.

The expected outcome of this testcase:

Because simple data transportation from the database to the caller application is the lion's share of the query, linear behavior is expected. A different outcome would mean that searching with a clause is an expensive operation.

The query in words: Simple scan with filter.
The query in SQL: SELECT * FROM Assets WHERE TrackNr=1

The results on the indexed database are visualized in figure 6.5.

The results on the database without an index are visualized in figure 6.6.

Observations and conclusions drawn from the results of this testcase:

No surprises here. BDB and SQLite 2 are the fastest DBMSs, all other DBMSs are a tad slower.

Again, MySQL demonstrates its use of cache during the warm run.

The difference in performance with and without index can be at least a factor two, as can be seen in graphs 6.5 and 6.6.

6.6 Testcase 4: simple scan with filter yielding no results

Purpose of this testcase:

Measuring the performance of an unsuccessful third use-case: Janet wants to search directly for a song title and play it.

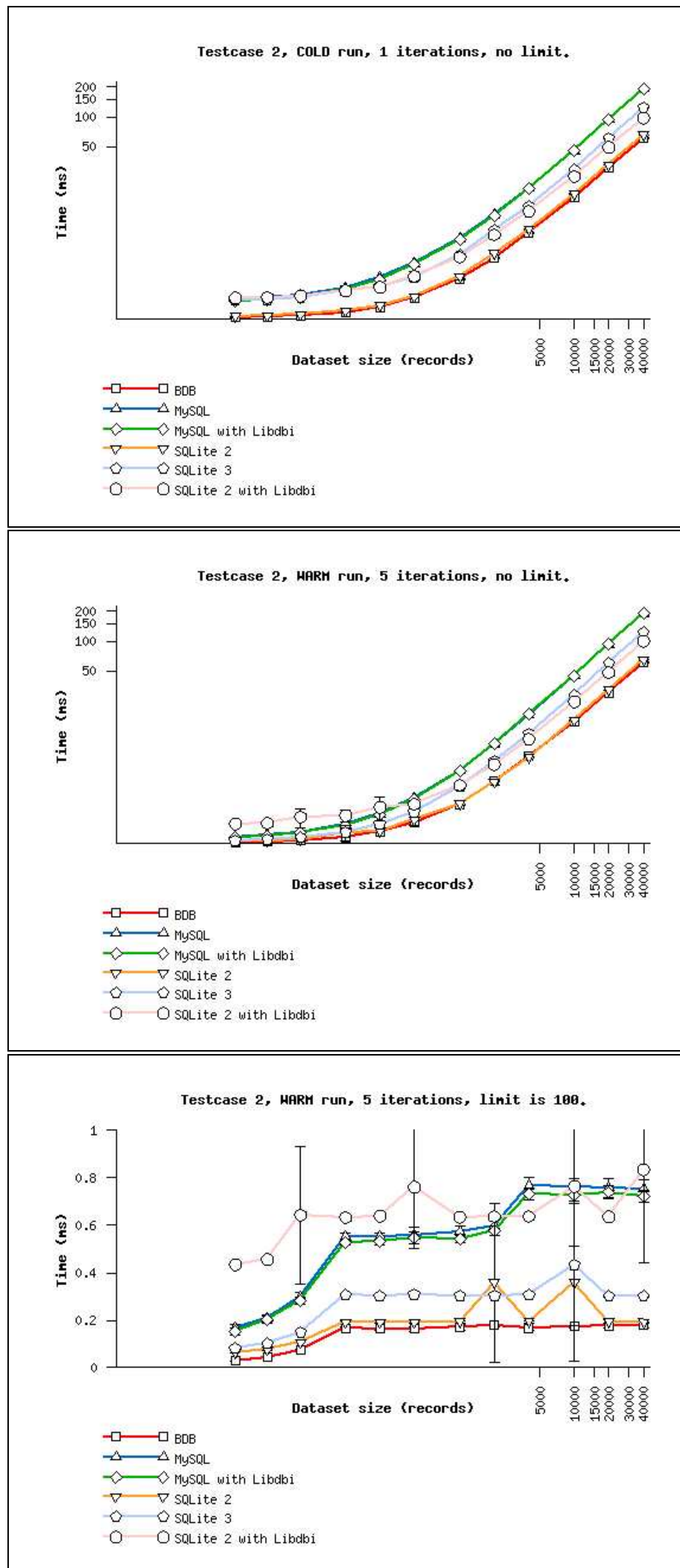

Figure 6.4: Analysis 1, case 2. Total time to retrieve all or some records from a table. Measured per dataset.

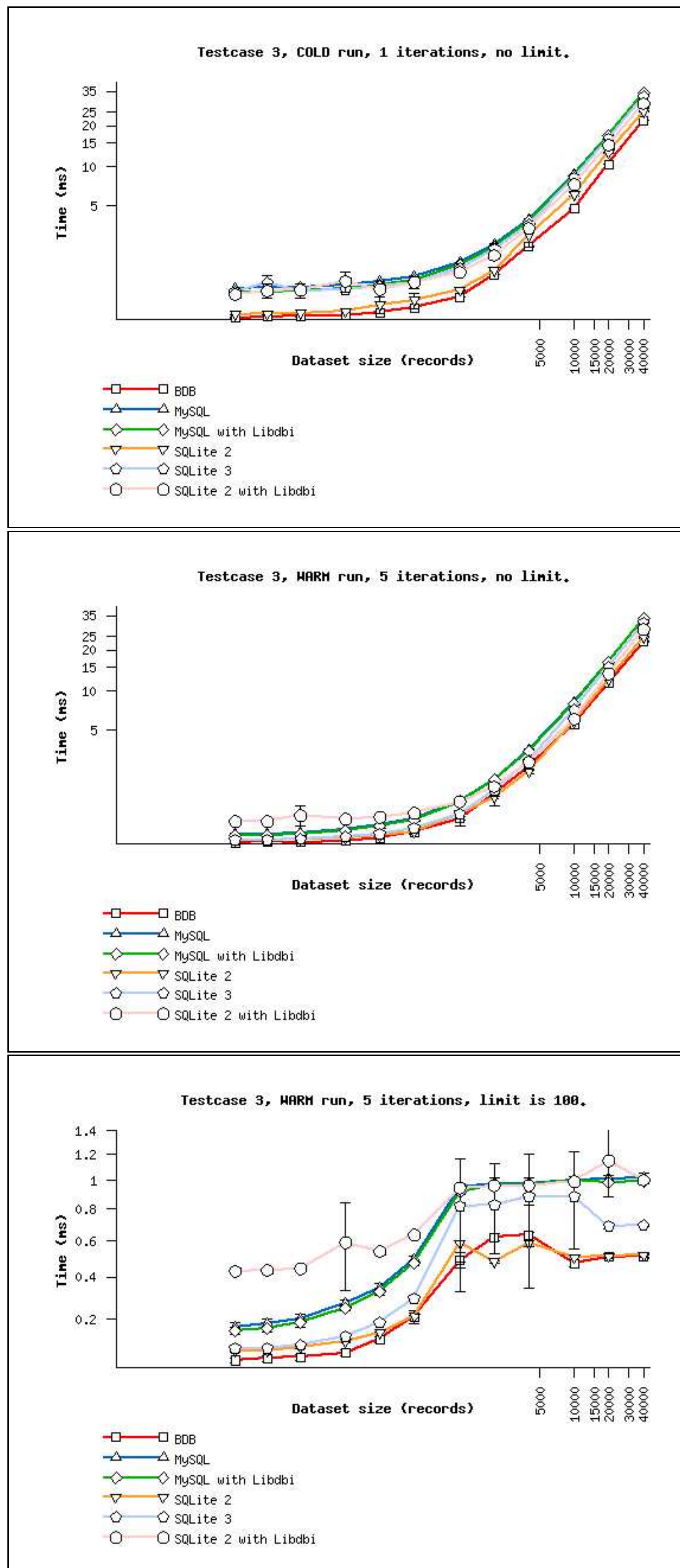

Figure 6.5: Analysis 1, case 3. Total time to search for records in an indexed table. Measured per dataset.

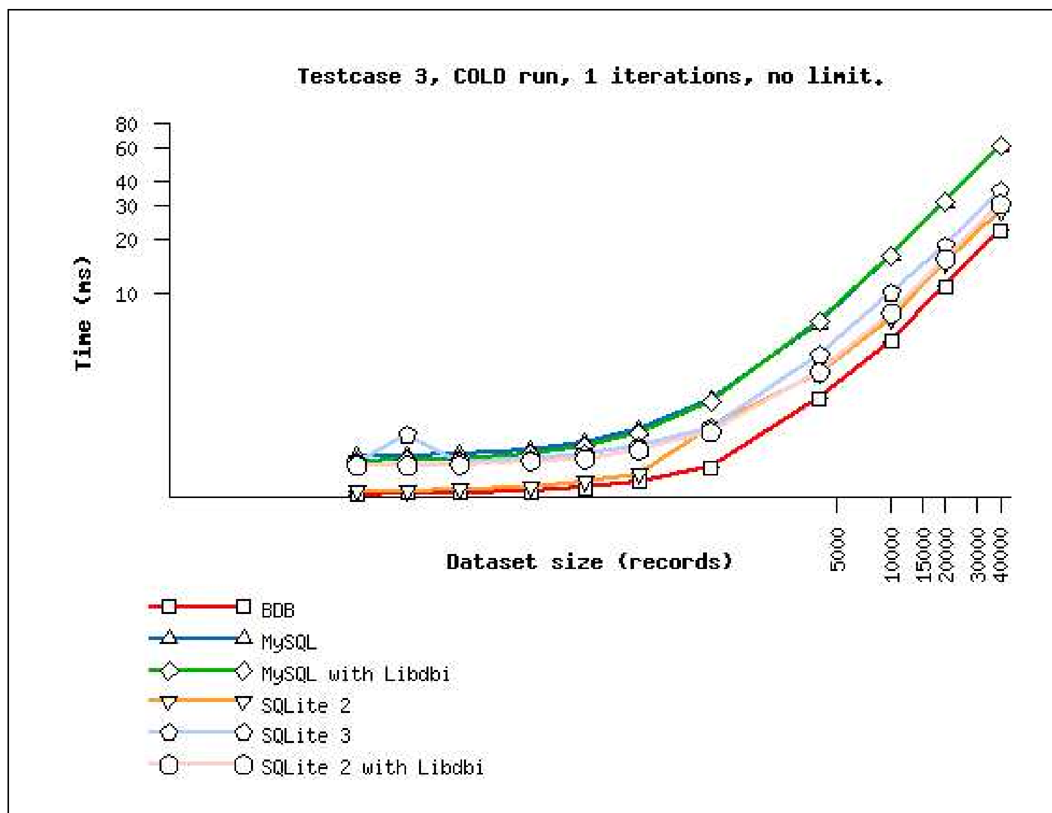

Figure 6.6: Analysis 1, case 3. Total time to search for records in an unindexed table. Measured per dataset.

The expected outcome of this testcase: This test activates the search in an index. The searched value is non-existent. Because there is no data transportation from the database to the caller application, logarithmic behavior is expected. A different outcome would mean that searching indexes is an expensive operation.

The query in words: Simple unsuccessful scan with filter.
The query in SQL: SELECT * FROM Assets WHERE TrackNr=9999999

The results are visualized in figure 6.7.

Observations and conclusions drawn from the results of this testcase:

All DBMSs are performing well, but MySQL and SQLite 3 suffer for a relative expensive initial cost.

The lines of BDB and SQLite 2 appear to be almost flat. It's because the index is a B-tree with a high fan-out, which keeps searches in big datasets efficient.

Compare the results of this testcase with the previous testcase and see that a simple search in itself is a cheap operation, it's the transport of data that is expensive. The logarithmic search operation is snowed under by the bulk of the operation, which is the default DBMS connection overhead.

The origins of the sudden speedup for MySQL around 400 records isn't perfectly clear. A call for help at the MySQL developer forum wasn't really helpful[11]. A clue might be found in the way MySQL works with indexes. It seems that the order of an index is not to be determined by the user. According to [BZ04], MySQL is able to figure out the best order by itself. MySQL will also traverse the index in the best order depending on the query. Even backwards when necessary. Searching an numerical ordered index backwards will instantaneously deliver the answer. The internals of the index mechanism will remain unknown as the source-code is not quite self-explanatory.

6.7 Testcase 5: inner-join

Purpose of this testcase:

Measuring the performance of the fifth use-case: Jill wants to see a list of available albums on her PIC. After selecting an album, she wants to see what songs are in it.

The expected outcome of this testcase:

Because of the raw data transportation from the database to the caller application, linear behavior is expected. A different outcome would mean that joining is an expensive operation.

The query in words: Join of 2 tables without a filter.
The query in SQL: SELECT alb.Title, ass.Title FROM Assets as ass, Albums as alb WHERE ass.AlbumId=alb.AlbumId

The results are visualized in figure 6.8.

Observations and conclusions drawn from the results of this testcase:

This is the first instance where BDB scales worse than any other DBMS, the join implementation may not be optimized. Everything is implemented in a rather straightforward manner as described in the BDB tutorial.

SQLite 2 is the fastest DBMSs.

[11] Posted at: `http://forums.mysql.com/read.php\?24,42330,42330\#msg-42330`

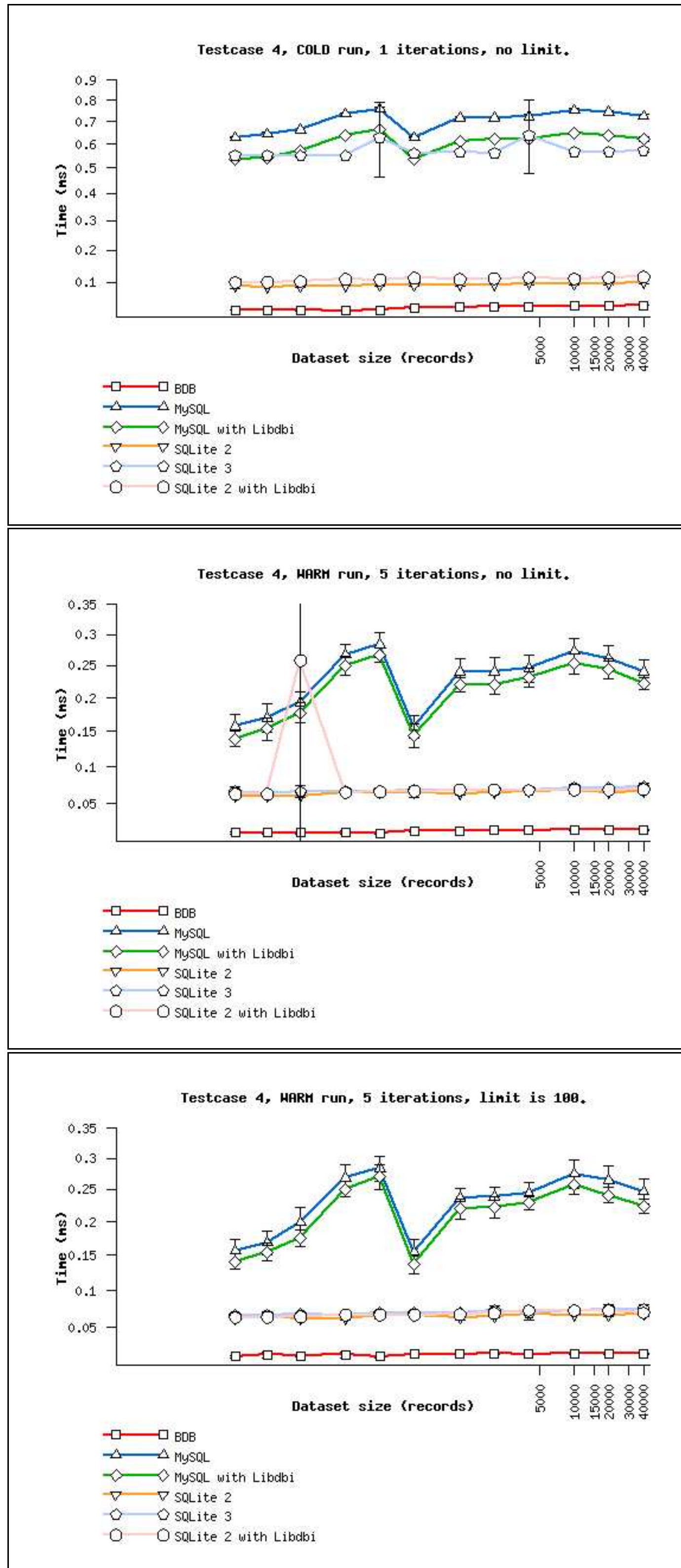

Figure 6.7: Analysis 1, case 4. Total time to search for non-existing records in a table. Measured per dataset.

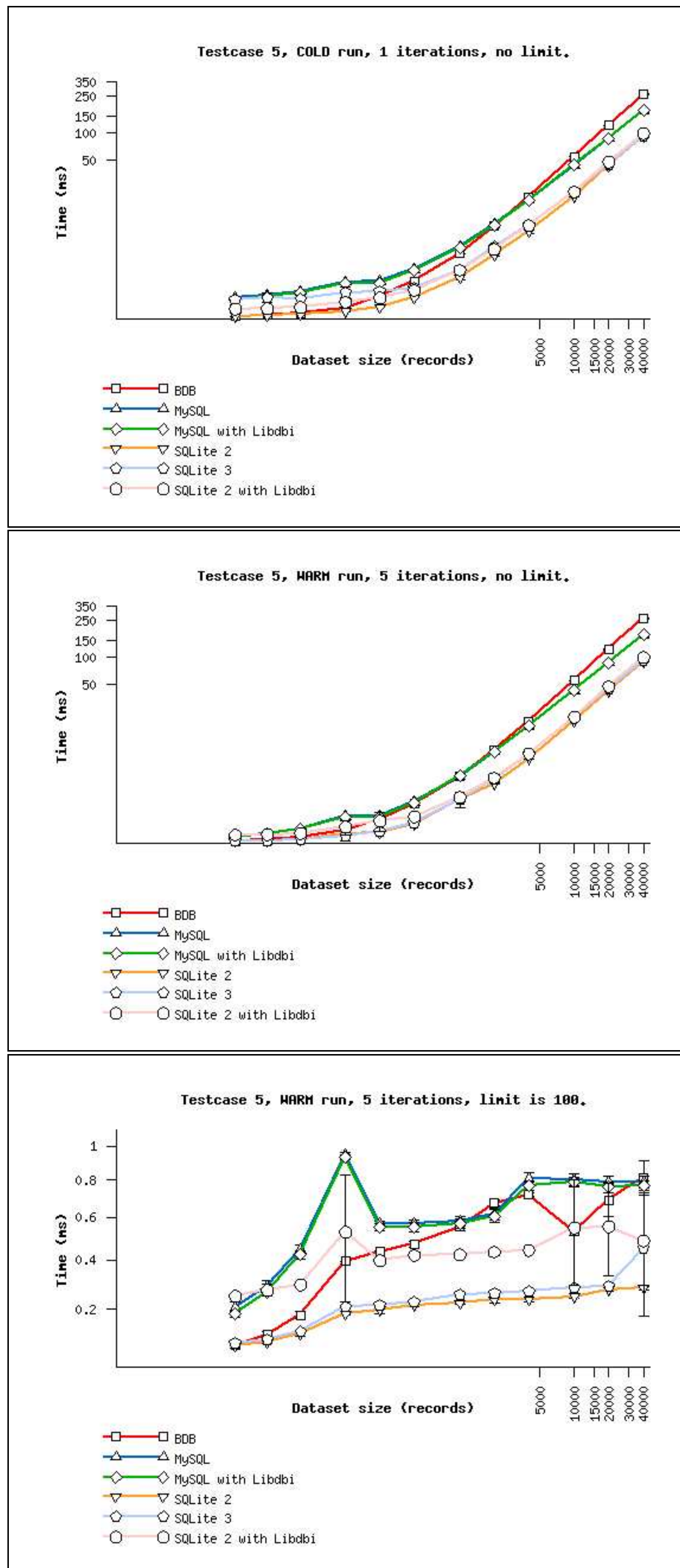

Figure 6.8: Analysis 1, case 5. Total time to retrieve all or some records from a join of two tables. Measured per dataset.

6.8 Testcase 6: inner-join with filter

Purpose of this testcase: Measuring the performance of the sixth use-case: Jill wants to see a list of available albums on her PIC. After selecting an album, she wants to see what songs are in it.

The expected outcome of this testcase:

Because of the raw data transportation from the database to the caller application, linear behavior is expected. A different outcome would mean that joining is an expensive operation.

The query in words: Join of 2 tables with one filter.
The query in SQL: SELECT alb.Title, ass.Title FROM Assets as ass, Albums as alb WHERE ass.AlbumId=alb.AlbumId and ass.TrackNr=1

The results are visualized in figure 6.9.

Observations and conclusions drawn from the results of this testcase:

Comparing the results with the previous testcase leads one to believe that the query optimizer is at work here. The filter on TrackNr is applied first to limit the number of records to join afterwards. This leads to better performance for the MySQL and SQLite DBMSs.

BDB performs quite bad. The reason is that the index search is not optimal. Because of implementation difficulty (lack of documentation and examples), the index is not made of integer values but strings. In the other DBMSs, the index is made of integer values which allow much quicker searches. It's probably not an BDB issue that can't be fixed after spending more time on the issue.

6.9 Testcase 7: simple scan with ordered results

Purpose of this testcase:

Measuring the performance of the seventh use-case: Jasper has a huge collection of music on his PIC. He wants a structured, sorted overview of all his music. He first wants to select an artist, then the album, then the song to play.

The expected outcome of this testcase:

Because the bulk of the operation is raw data transportation from the database to the caller application, linear behavior is expected plus an additional component of $O(nlogn)$ (on average) for sorting the results. A different outcome would mean that sorting is an expensive operation.

The query in words: Simple scan with sorting.
The query in SQL: SELECT * FROM Assets ORDER BY Title

The results are visualized in figure 6.10.

Observations and conclusions drawn from the results of this testcase:

The SQLite variants and especially version 2 are the best overall performers. BDB is just as fast on small datasets but becomes slower at bigger datasets.

MySQL is the slowest, but seems to demonstrate good performance through caching at small datasets.

6.10 Aggregated results

In this chapter, 7 testcases have been benchmarked. To appoint the DBMS that performs best in all testcases combined, the testcases have to be rated.

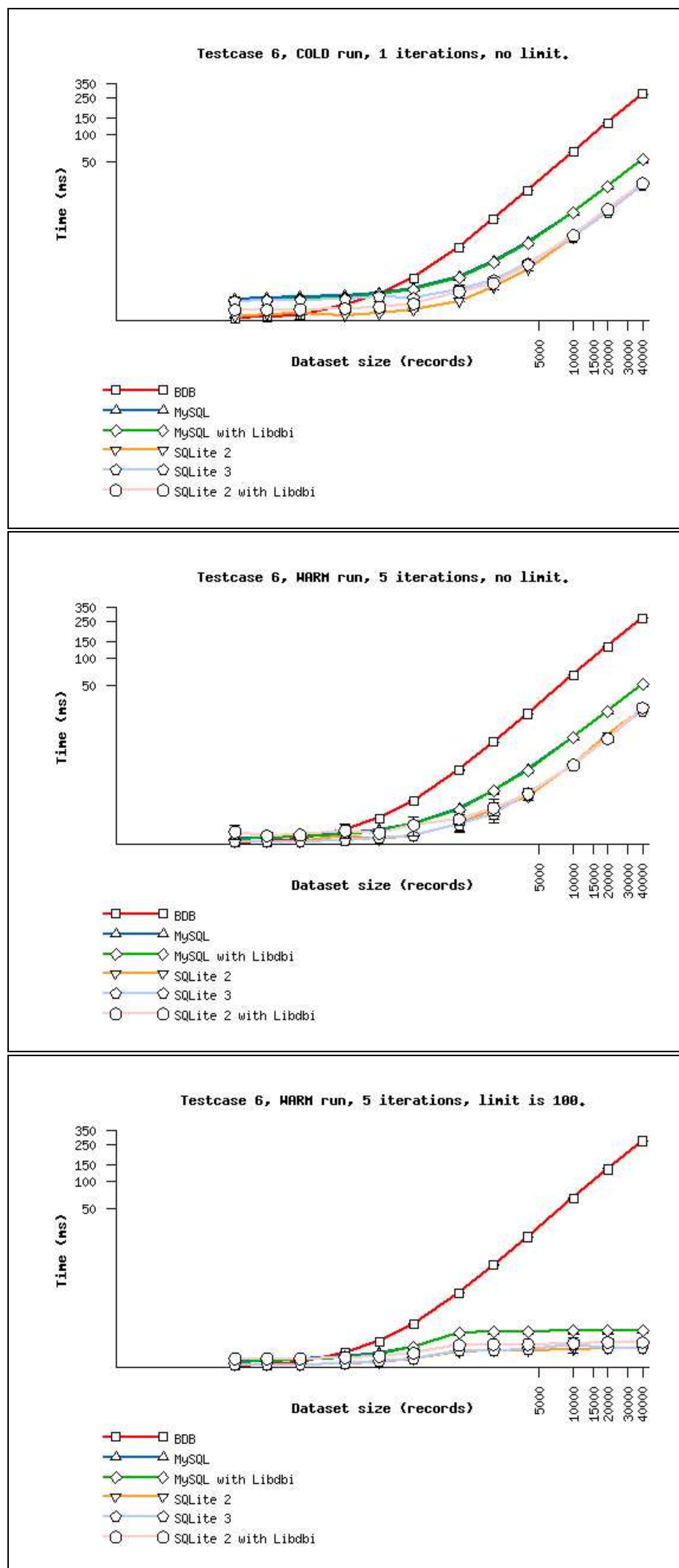

Figure 6.9: Analysis 1, case 6. Total time to retrieve all or some records from a join of two tables including one filter. Measured per dataset.

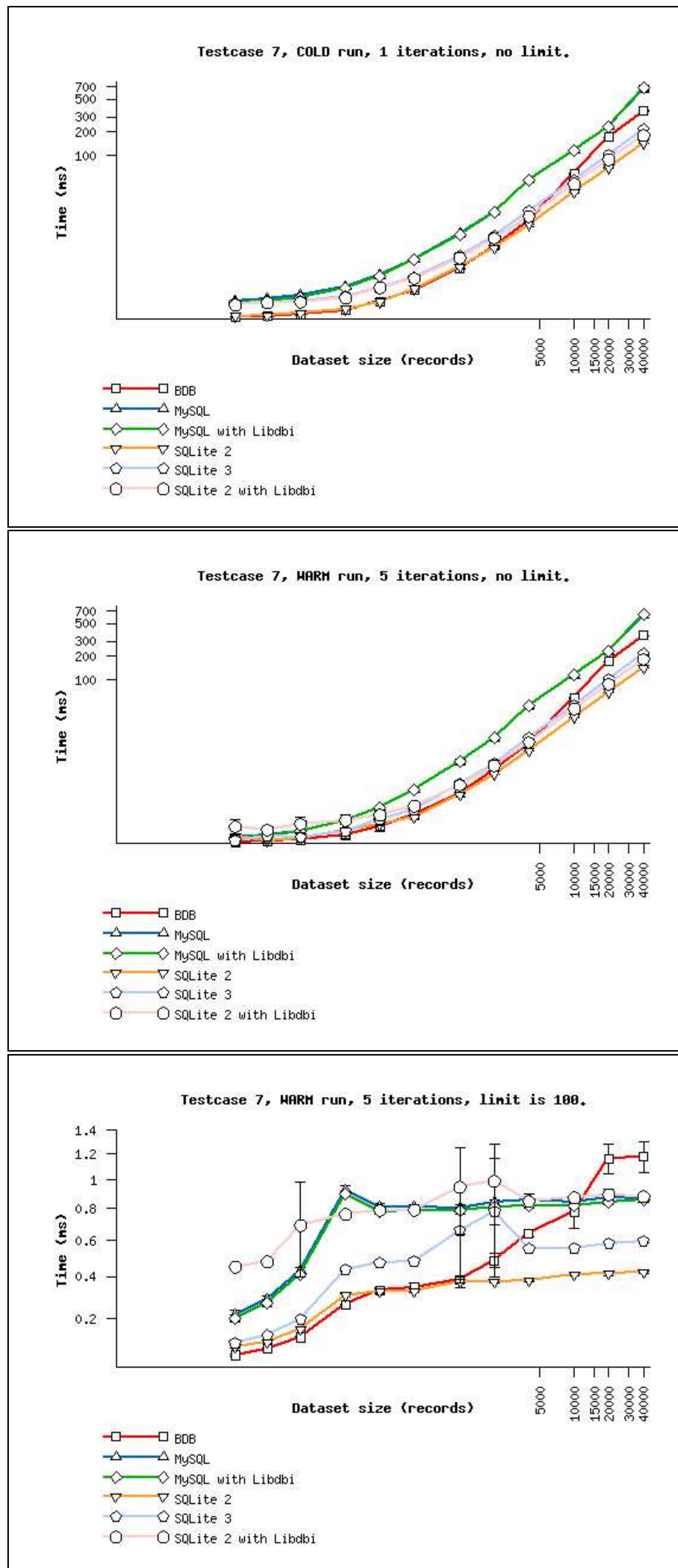

Figure 6.10: Analysis 1, case 7. Total time to retrieve all or some records from a table, sorted alphabetically. Measured per dataset.

Test-case(Use-case)	Target 1 (weight)	Target 2 (weight)	Target 3 (weight)	All targets (weight)
1(1)	5	5	20	30
2(2)	10	10	40	60
3(3)	20	20	3	43
4(3)	20	20	2	42
5(4)	20	20	2	42
6(4)	20	20	3	43
7(5)	5	5	30	40
	100	**100**	**100**	**300**

Table 6.1: Frequency table of test-cases for each target platform.

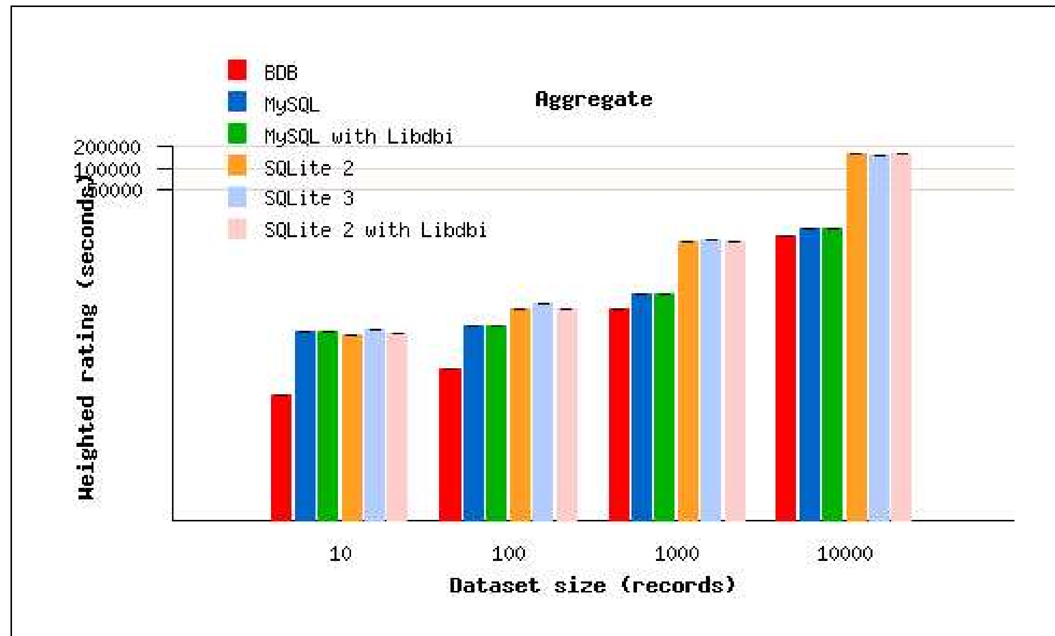

Figure 6.11: Initial aggregated results. Total time to execute all tests (cold) a certain number of times, using the frequencies of target 3. Broken down per dataset. Lower is better.

Therefore, from the use-case frequency table 2.1, a test-case frequency table is derived that is shown in table 6.1. Because a use-case can have more than one outcome, use-case 3 is split out in test-cases 3 and 4 and use-case 4 is split out in test-case 5 and 6. This way, all common situations will be benchmarked.

Applying formula 2.1 to the test results, the initial score is displayed in figure 6.11. See chapter 7 for the final score, after the DBMSs have been tuned to perform optimal.

At this moment, BDB is the all-round winner. Because no optimization is done yet, BDB wins much time on testcase 1, while it performs the worst on joins. BDB is closely followed by MySQL. SQLites negative results are mostly influenced by its slow insert times.

Chapter 7

Benchmark result analysis 2

In this chapter, the several kinds of benchmarks are run and analyzed.

First, several measurements are performed that really are a continuation of chapter 6. The same measurements as in the last chapter are run again on a different target platform. Also, where in the last chapter MySQL was benchmarked with the default database engine, other engines are now benchmarked in a quest for the optimal engine.

Next, a new topic is benchmarked: the overhead of setting up a connection per query versus one connection for many queries.

7.1 Benchmarking on the ARM architecture

Hardware platform:

Target platform 2, the mobile audio/video player based on the ARM[1] architecture is benchmarked.

DBMSs:

In this run, the following DBMSs are benchmarked:

- SQLite v2
- SQLite v2 with Libdbi

Only one day was put aside for porting the benchmarking framework as the internship had already officially ended. SQLite was the only DBMS that compiled on the platform within the time limit.

Schema:

The schema remains the same one as used in chapter 6, the second benchmark run (see figure D.1).

Datasets:

The dataset sizes used in the benchmark are: 10, 20, 40, 100, 400 and 1000 assets. More and bigger datasets would delay the benchmark too much.

Dimensions: The same type of graphs are produced as in chapter 6.

[1] The company "Advanced RISC Machines Ltd." ships ARM (embedded RISC) processors and architectures

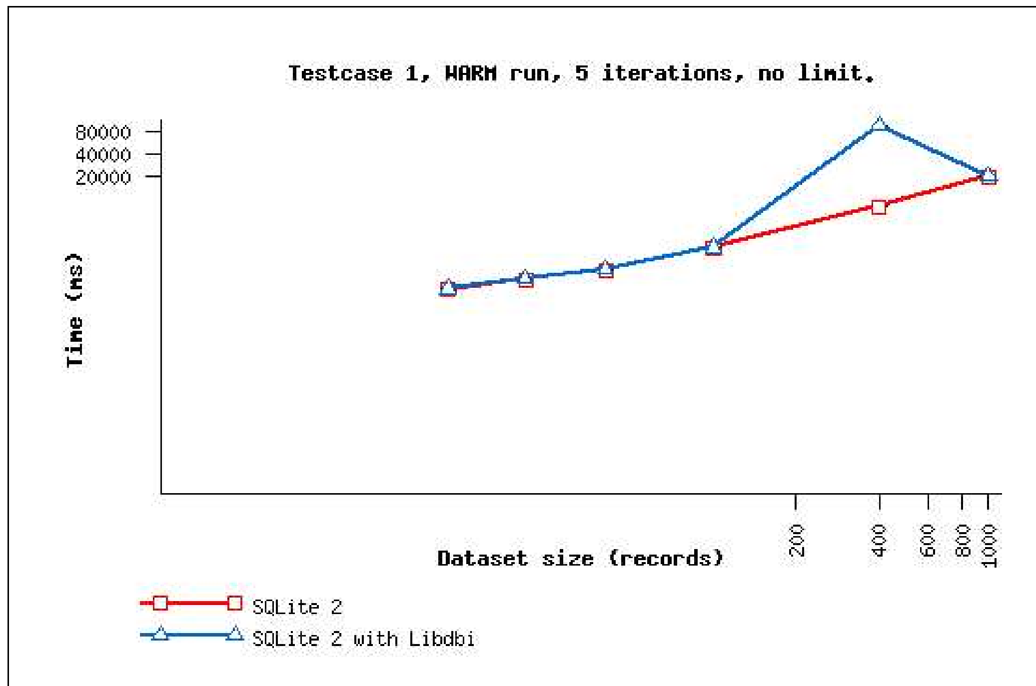

Figure 7.1: Analysis 2, case 1. Total measured INSERT time per dataset in a fresh database.

Iterations: Warm run, unlimited results, 5 iterations.

7.1.1 Observations and conclusions drawn from the results

As the purpose and expected outcome of the these testcases are mainly similar as in chapter 6, only the interesting details will be described. All results are visualized in figure 7.1 to figure 7.7.

In testcase 1, SQLite shows a glitch when inserting 400 records. Unfortunately, the measurement can't be repeated. It can have several causes. The filesystem on the device was quite full, thus an unfortunate (non-linear) range of blocks could have been assigned for the database. Unlike MySQL, SQLite only uses one indexing scheme, so the cause is probably outside SQLite.
Compared to the performance on target platform 1 as measured in chapter 6, the results are of a similar shape, except for a scaling factor.

The amount of overhead in data transport from the DBMS to the user application is substantial. If that transport has to go through Libdbi, then the performance is even half of the performance without wrapper. Testcase 2 (also 3 and 5) is a good example of this behavior. On all these queries on target platform 2, SQLite without wrapper is at least 25 times slower as on target platform 3.

In testcase 4, there's no data transport, just some work performed by the DBMS. Now Libdbi doesn't have any real performance tax. Although it's around 400 times slower as on target platform 3.

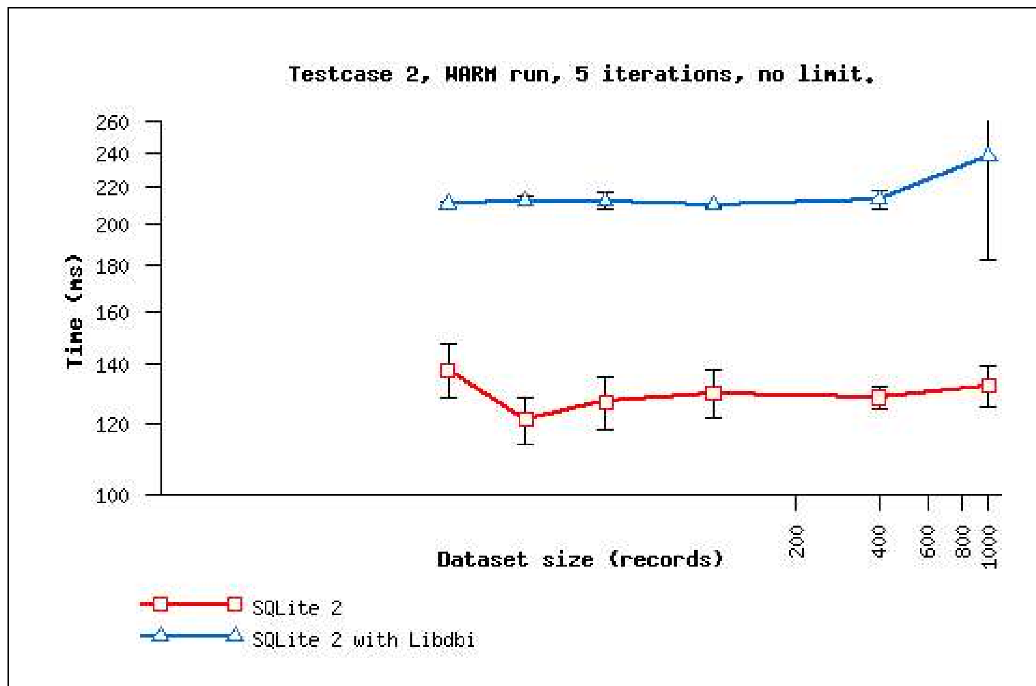

Figure 7.2: Analysis 2, case 2. Total time to retrieve all or some records from a table. Measured per dataset.

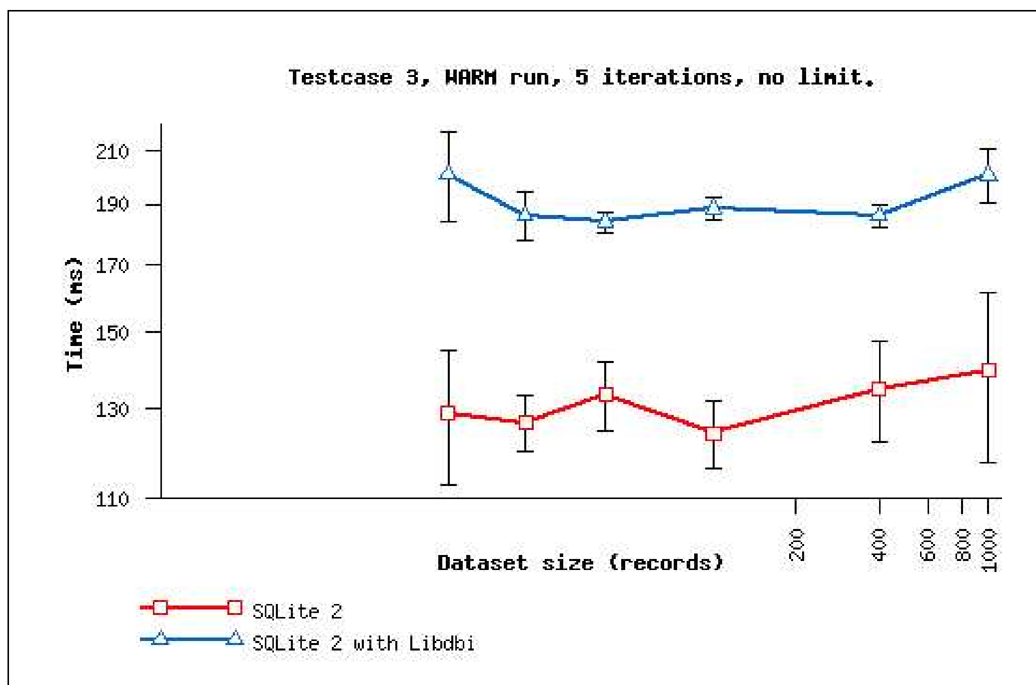

Figure 7.3: Analysis 2, case 3. Total time to search for records in a table. Measured per dataset.

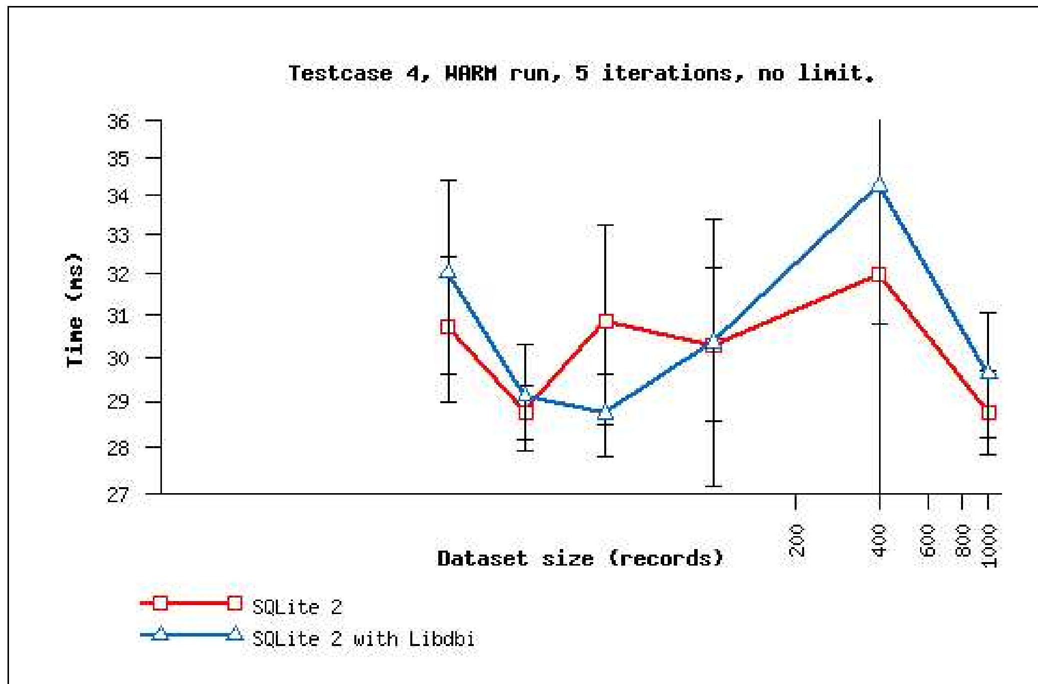

Figure 7.4: Analysis 2, case 4. Total time to search for non-existing records in a table. Measured per dataset.

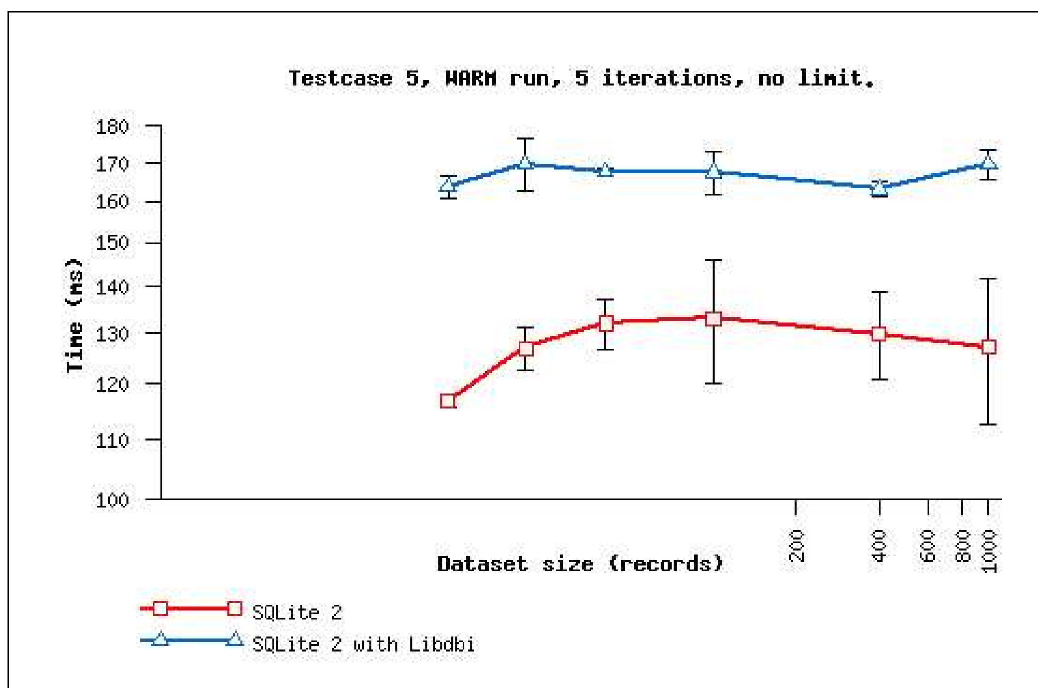

Figure 7.5: Analysis 2, case 5. Total time to retrieve all or some records from a join of two tables. Measured per dataset.

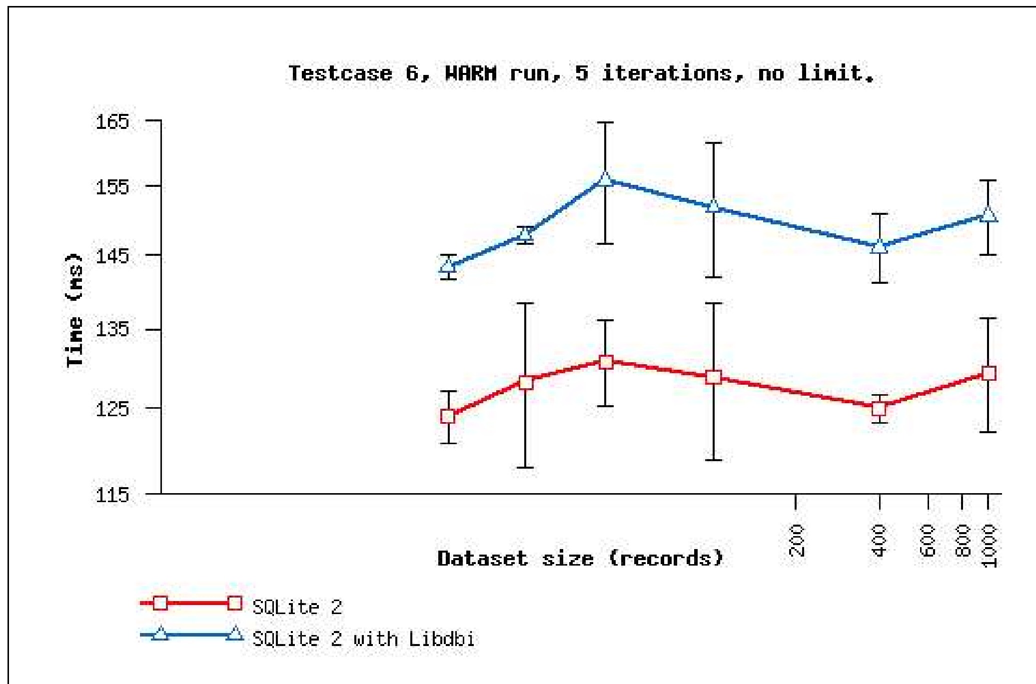

Figure 7.6: Analysis 2, case 6. Total time to retrieve all or some records from a join of two tables including one filter. Measured per dataset.

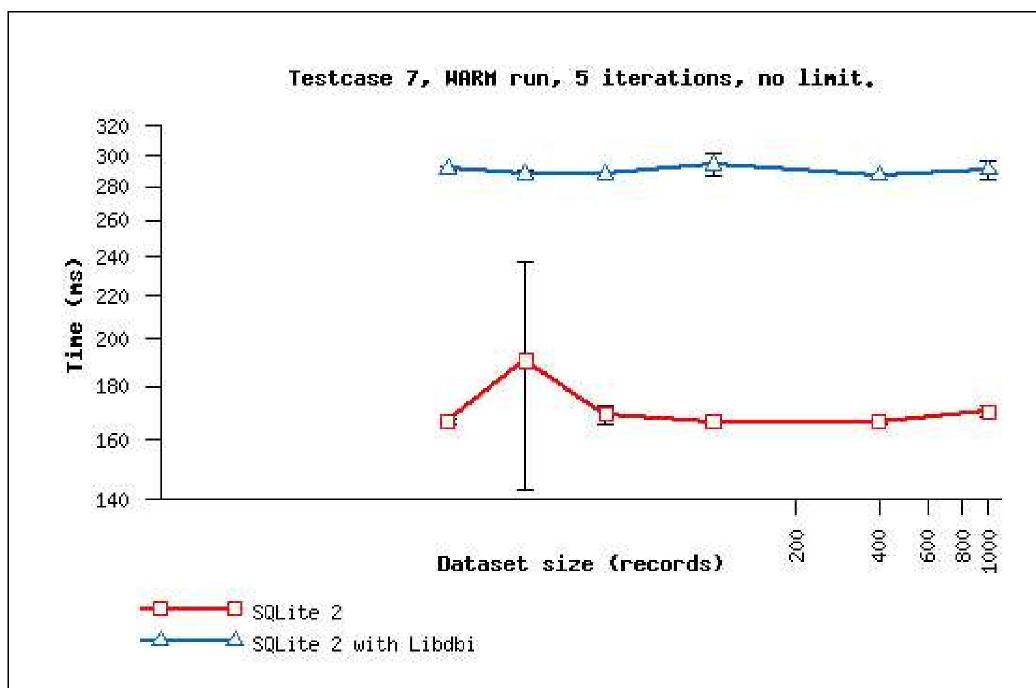

Figure 7.7: Analysis 2, case 7. Total time to retrieve all or some records from a table, sorted alphabetically. Measured per dataset.

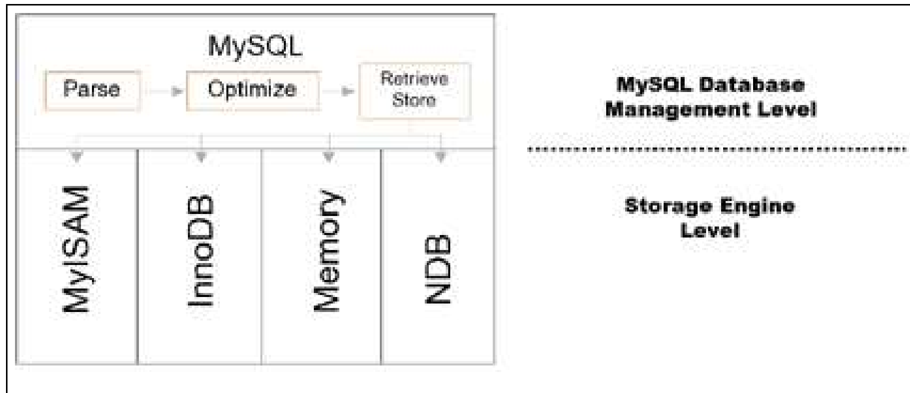

Figure 7.8: The MySQL architecture

	MyISAM	InnoDB	MEMORY	NDB
Transactions	no	yes	no	yes
Foreign keys	no	yes	no	no
Locking level	table	row	table	row
BTREE indexes	yes	yes	no	yes
FULLTEXT indexes	yes	no	no	no
HASH look-ups	no	yes	yes	yes
Relative disk use	low	high	none	low
Relative memory use	low	high	low	high
Target platform 1 and 2 suitability	high	low	medium	low
Target platform 3 suitability	low	very high	high	medium

Table 7.1: MySQL storage engine architectures compared. A suitability rating is added for the target platform. It's based on the features in the table.

With the exception being testcase 4, all testcases exceed the 100ms threshold that was defined in section 2.3 that users still perceive as an instant response.

7.2 Database engines for MySQL

MySQL supports several storage engine architectures[2] as visualized in diagram 7.8, they're compared in table 7.1 and have been benchmarked on target platform 3, the home server. There was no time left to perform a benchmark on the other target platforms.

MyISAM aims for low overhead, so its fast but doesn't support transactions nor foreign keys. At least, so it is advertised. MyISAM is benchmarked in the same way as InnoDB was benchmarked in chapter 6. On all testcases but the last one, they performed quite similar.

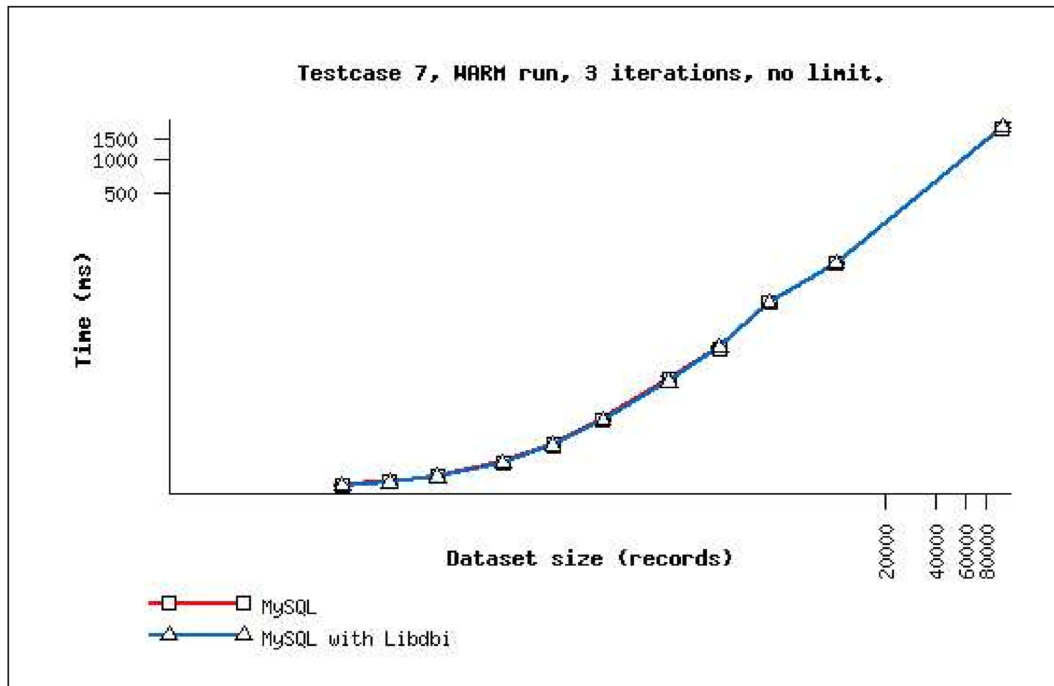

Figure 7.9: Analysis 2, case 7, MyISAM engine. Total time to retrieve all or some records from a table, sorted alphabetically. Measured per dataset.

InnoDB offers features like transactions and foreign keys, but they come at a cost. The trade-off is memory and disk space usage that is 3x as high compared to MyISAM. InnoDB does not require any locking for a SELECT. This makes extremely high concurrency possible. InnoDB is the default MySQL engine and was therefore used during the benchmarks in chapter 6.

The order operation takes at least two times more time than with InnoDB as can be seen in figure 7.9. The explanation is hidden in the details of MyISAM. With MyISAM, string indexes are space compressed[3]. Space compression makes the index file smaller if the string column has a lot of trailing space or is a VARCHAR column that is not always used to the full length. Prefix compression is used on keys that start with a string. Prefix compression helps if there are many strings with an identical prefix. It's only logical that this space-saving results in a decreased speed because of the time needed for decompression.

The **MEMORY** engine (formerly called HEAP) keeps all table data in memory. It's fast but the trade-off here is a high risk of data loss and it requires more memory than disk-based tables. The data loss risk shouldn't always be an issue: if a copy is kept on disk and memory is used for storing the read-only database.

Like MyISAM, the MEMORY engine is also benchmarked. The MEMORY engine performed blazing fast in all testcases. As the results couldn't be measured with great accuracy, because of the fast speed, the standard deviation is big. Only testcase 4 (no particular reason for this testcase) is shown in figure 7.10 as all testcases resulted in likewise graphs.

Note that the amount of assets is at its maximum before going out of memory on target platform 3 (home server). Discarding the indexes will provide some extra space at the expense of speed.

NDB, the MySQL cluster storage engine is not considered as there's no cluster available to perform the benchmarks on. Also, consumers aren't expected to use expensive clusters to look up a song.

[2]More details at: `http://dev.mysql.com/tech-resources/articles/storage-engine/part_3.html`

[3]More details at: `http://www.php-editors.com/mysql_manual/p_manual_Table_types.html`

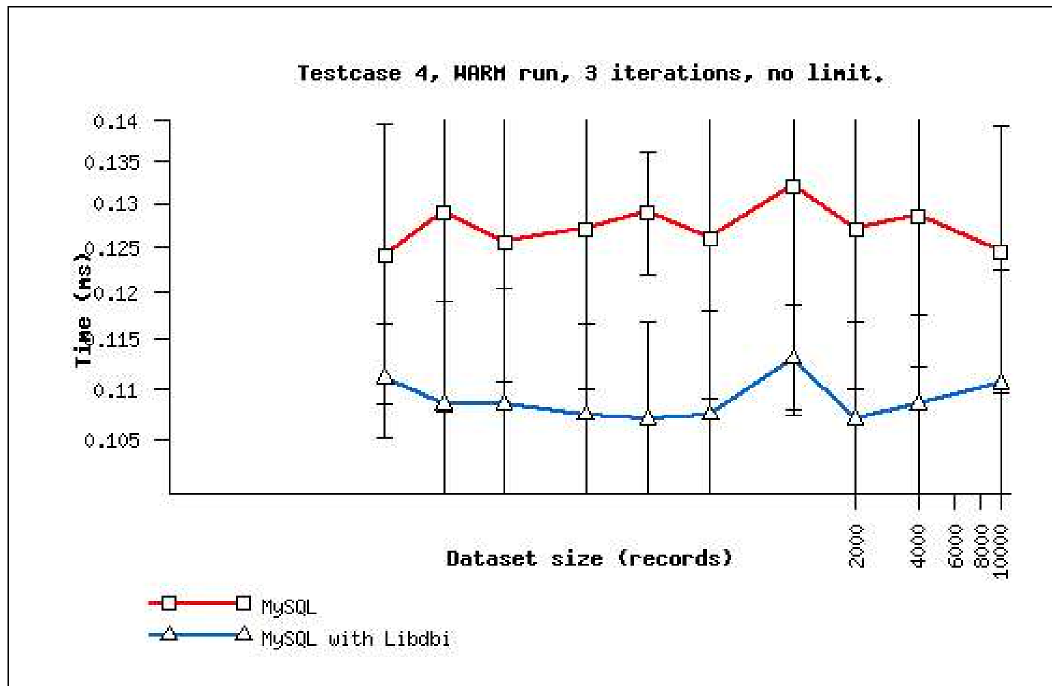

Figure 7.10: Analysis 2, case 4, MEMORY engine. Total time to search for non-existing records in a table. Measured per dataset.

	SQLite 1st	≥2nd	MySQL 1st	≥2nd
Connect	0.4	0.1	28	1
Query execution	1.6	1	30	2
Disconnect	0	0	0	0

Table 7.2: Connect, query execution and disconnect duration (ms)

Combining MyISAM or InnoDB with MEMORY could result in a feasible solution for the mobile targets. In this scenario, reduced energy usage and a speedup is gained as long as the user uses the device mostly in a read-only fashion. To perform high speed operations on a fairly limited amount of data, a temporal MEMORY table has to be created and data from a disk-based table should be copied to it as follows: *CREATE TABLE memtable ENGINE=MEMORY SELECT * FROM disktable;*. On writing, both tables should be updated. A simple wrapper could take care of this.

7.3 Connection overhead

Creating a connection from the user application to the DBMS is not a frequent operation, so it hasn't been benchmarked before. However, in MDB, sometimes a new connection is opened and closed thereafter for just a few query. By using only one connection (or a connection pool, if one is not enough) that isn't closed after one query execution, MDB will be a bit faster. Especially if the DBMS server and client are not located on the same device. It's useful to know at least something about the connection setup and disconnect costs. Therefore, a little benchmark is created.

See table 7.2 for the measurement results. Only the time of creating a connection for the second (and later) time is relevant because the connection setup overhead would only matter when it happens frequently. SQLite is about 10 times faster than MySQL.

The reasons for this behavior are clear. MySQL uses a "heavy" client-server architecture where not only the client library is activated, but also an internal connection to the server has to be established through a socket. Then the server needs to handle this incoming connection. SQLite is more lightweight as the DBMS really is only a client library already compiled in with the caller application.

Most queries in MDB are relative simple data lookups, where the overhead of the connection setup and teardown is not negligible.

Chapter 8

Results and conclusions

Looking back on the requirements (chapter 2), several conclusions have been found during this graduation project and will be explained in this chapter.

Note that the benchmarking framework mainly tests DBMSs on performance and scalability, that's what these conclusions will be based on. The benchmarking framework is not yet advanced enough to take any energy and memory measurements into account.

The first and foremost conclusion is that one "complete" solution satisfying all requirements doesn't exist. Despite all query optimizers, caches and other trickery seen and tried, there is no silver bullet! The problem lies in the trade-offs or compromises that are (seemingly) unavoidable and depend highly on the target platform.

No **ease of use** for the programmer without a **performance** penalty. Outstanding performance can only be gained if the database and queries are sufficiently tuned. Especially with big joins, a query optimizer really needs extra information from the programmer to perform well (see chapter 3).

The Libdbi DBMS abstraction layer, that's currently used in prototype products, is adding complexity and a performance penalty where there's really no problem to solve. The problem that is intended to be solved using Libdbi is being future proof. "Switching from one RDBMS to another should take only a fraction of the time it would otherwise have taken without Libdbi", such is the idea behind Libdbi. The issue with Libdbi is that the benchmarks on the target platform 2 (see section 7.1) sometimes show a factor two performance degradation when using Libdbi. It's a performance bottleneck, so the situation has to be resolved. Fortunately, it's very easy. What all benchmarked DBMSs have in common is basic SQL support, which is really all that's being used in the prototype. If developers write valid, standard SQL queries [Kli04], only the connect, disconnect and query mechanism have to be customized per DBMS. To support multiple DBMSs simultaneously, it can be abstracted away. This is so easy, it's the way the benchmarking framework is implemented itself.

A DBMS abstraction layer would only be needed if it would emulate functionality an DBMS would be missing.

With this being said, lets look at the available results.

8.1 DBMS conclusions

First, the benchmarked DBMSs receive a verdict. Then a DBMS recommendation per target platform is given.

BDB is just the data storage layer, it does not support SQL or schemas. In spite of this, BDB is twice the size of SQLite, the other embedded DBMS. A comparison between BDB and SQLite is similar to a comparison between assembly language and a dynamic language like Python. BDB is probably much faster in every situation as long as the code is carefully crafted. But it is much more difficult to use and considerably less flexible. On the other hand (see section 3.3.3), queries formulated using an SQL query language contain little useful information for estimating query performance. Internal knowledge of the database structure, data distribution and query optimizing strategy are necessary to develop effective query statements.

SQLite is a fast, small and simple DBMS. It does what is advertised, but nothing more. Nonetheless, it offers all the features required by the PIC project. This DBMS strikes a good balance between performance and ease of use.

MySQL is a popular, stable DBMS. It's not bad at all, but it doesn't really have any advantages over SQLite that would justify it's slower speed. May the situation ever arise that a DBMS requires an internal modification or optimization, the SQLite source code is extremely accessible. Whereas the MySQL source code is more complex and (at least initially) harder to read.

8.1.1 DBMS recommendations per target platform

Target platform 1 and 2: BDB (note that this is a prediction because only SQLite has been benchmarked on an embedded platform). The limited amount of queries required on such a platform make this the excellent platform to optimize the queries by hand using BDB and save resources.

Target platform 3: SQLite version 3. Because it's a fast, lightweight RDBMS that supports features like concurrency and transactions for a home server. In contrast with client-server RDBMSs like MySQL, there's no permission control on the DBMS level. This should be handled elsewhere.

All platforms combined: SQLite. If there's one DBMS to choose to cater for all platforms, it should by SQLite. It offers the ease of SQL while still being a small, fast, embedded DBMS that supports most (all?) necessary features.

Please continue reading the benchmarking framework recommendations in chapter 9.

Chapter 9

Future work and recommendations

9.1 Introduction

During the early formulation of requirements until the analysis of benchmark results in the end, many ideas popped up, insight was gained and situations changed. Sometimes, anything new was simply discarded, sometimes it was pushed aside temporarily and written on a to-do list. Sometimes it even changed the direction of the project.

Due to a myriad of reasons, new ideas were often postponed, mostly because of time constraints. Hopefully, someone will follow up on this project so it will advance further. Therefore, the remainder of this chapter lays out the various directions in which the framework could be advanced. The framework could be made more universal, more complete and more accurate. The last section contains recommendations to continue work that will have an attractive yield to time ratio.

9.2 Towards a more universal benchmarking framework

The benchmarking framework can be expanded in several new and interesting directions. One direction is turning the framework into an even more abstract framework by becoming client programming language independent.

9.2.1 Client programming language independent benchmarking

Performance of a database application is determined by the efficiency of both the client and the server. Naturally, the client performance is affected by the programming language in which it is written. This not only refers to DBMSs like MySQL that use a client-server architecture, but also to embedded DBMSs that have multiple "client" APIs.

In [Pac03], benchmarks are performed against MySQL using clients in several programming languages. In some tests, the Java client outperformed the C client by a factor two. The client may have more impact than expected. The difference in performance is also seen in chapter 6 in the testcases with Libdbi versus the native clients without Libdbi.

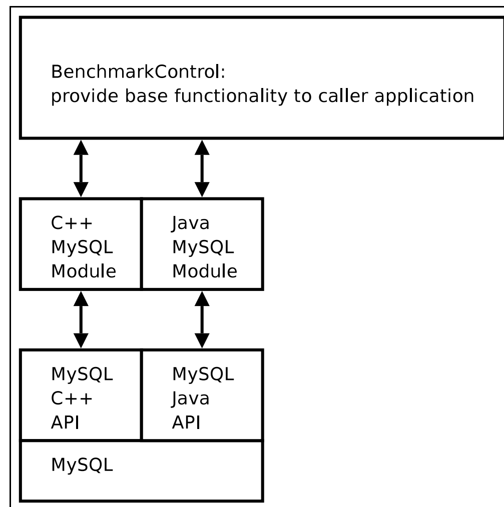

Figure 9.1: Proposed new benchmarking framework modularization.

In the current framework, only C++ DBMS APIs are used for benchmarking because the benchmarking framework itself is written in C++. The applications where the benchmark framework was written for, are and will also in the future be written in C++. A more general framework should loose the C++ requirements for DBMS APIs by delegating the core API functionality (connect, perform query, disconnect) to other subsystems that may be written in other programming languages. This requires further modularization of the framework that is currently designed as in figure 5.2. The benchmark control component should not directly communicate with underlying DBMS APIs, but should instead delegate this task to specialized modules that are programmed in no specific language. The new BenchmarkControl still receives benchmark parameters from the caller application (overseer.py) and will now delegate the actual benchmark to the specialized modules, through pipes or IPC[1]. The design is visualized in figure 9.1.

9.3 Towards a more complete benchmarking framework

Even though the benchmarking framework is quite complete for basic performance testing, creating more testcases will broaden the test coverage. Measuring more dimensions than time would also be very useful, although more difficult. Finally, new environments to use the benchmarking framework in are discussed in this section.

9.3.1 Supporting more operations

The testcases performed in this thesis consist of data insertions (INSERT in SQL) or search operations (SELECT, WHERE and JOIN in SQL). Because updates (UPDATE in SQL) and deletes (DELETE in SQL) are the least frequently used operations, they're currently not benchmarked. It's not hard to add testcases to benchmark those operations, but due to time constraints, higher quality of a small number of benchmarks was preferred over a larger quantity to achieve broader coverage (more completeness) of the benchmarks.

9.3.2 Measuring more dimensions

Only one dimension is currently measured in the current benchmarking framework: time. Benchmarking several other dimensions would be helpful, although technically more difficult to accomplish.

[1] Inter-process communication, see `http://en.wikipedia.org/wiki/Inter-Process_Communication`

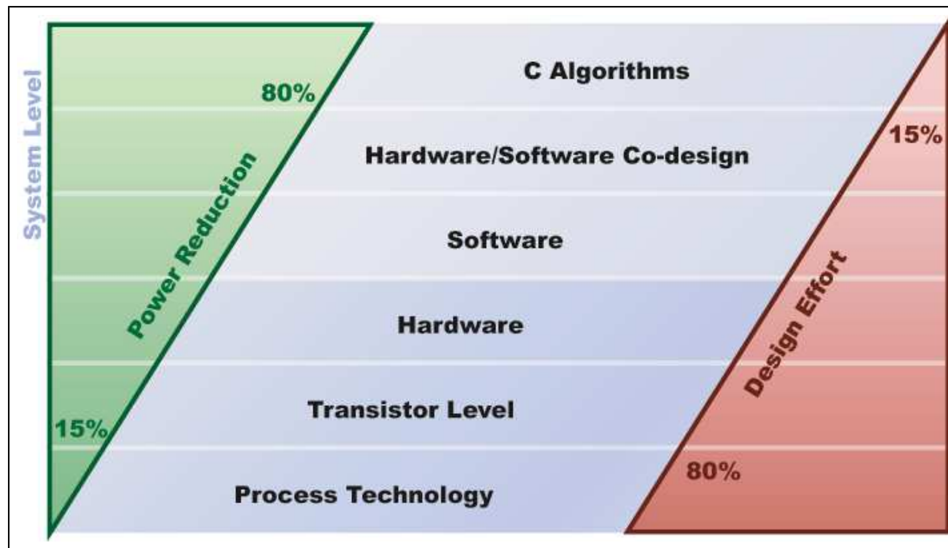

Figure 9.2: Power reduction versus design effort.

Energy consumption

Especially for target platforms 1 and 2, knowing more about energy usage is valuable. Having measurements with this dimension available per testcase may very well result in a different conclusion in the DBMS comparisons of chapter 6 where time is the only criterion.

Unfortunately, it's very difficult to perform these measurements. There are three options: combining measurements with a theoretical model, hardware simulation in software or measuring with hardware.

Combining measurements with a theoretical model

This first option is a free but rather complex solution using only software. Only the harddisks energy consumption is computed. The author of [Mes03] has access to a Linux kernel patch that measures and logs the amount of disk accesses performed by the kernel. Combining the timestamps in these logs with the timestamps generated by the benchmarking framework, the amount of disk accesses triggered by a benchmark can be found. This number is to be fed into a theoretical model that also incorporates the harddisk properties (found on the harddisk data-sheet) to compute the total amount of energy consumed. The model is quite accurate. It also includes possible spin-up events.

Hardware simulation in software:

The most expensive option, but one that requires the least effort is using simulation software from PowerEscape Inc.[2]. By adding only a few extra lines of code to the benchmarking application and the DBMSs and recompiling those, a detailed profile is provided in which can be seen how much energy is consumed in each function. According to PowerEscape Inc., optimizing the software yields the most savings in power, see figure 9.2.

Hardware simulation in software The last option left is doing the entire measurement in hardware. It's the most complete and accurate option, but requires dedicated hardware that has connectors available for connecting probes. Philips has such an energy consumption monitor. Unfortunately, there was no time left to actually use it.

[2]Company website: `http://www.powerescape.com`

Memory usage of DBMSs while executing queries: This feature has been on the radar for a while. Unfortunately, it proved too difficult to accomplish. MySQL for example, has one or more server processes running to handle query executions. One can retrieve the memory usage status from the proc[3] file system for example, but during the time the query is executed, new memory is allocated and discarded memory deallocated. The only way to gain enough information is to poll for this information very often. Polling will influence the measurements, so it's not an accurate solution. Even after discussing this issue with colleagues, a solution is yet to be found.

Memory usage of database on harddisk:

This feature has been partially implemented in the framework. Hooks are in place, but due to time constraints, the complete implementation has never been finished.

9.3.3 Measuring more environments

The benchmarking framework is currently very flexible. Many variables can be set to control a benchmark, see chapter 5. However, there are also "variables" that do not belong to the framework itself but are nevertheless interesting.

Different file systems

Current generation MP3 players (like platform target 1) are so called USB mass storage devices. The USB standard does not specify which file system should be used on such devices; instead, it mainly provides a way of reading out sectors as on any harddisk device. Operating systems are free to format this storage area with any file system. But due to the consumer market dominance of Microsoft Windows, the most common file system on embedded devices such as sticks, cameras or MP3 players is the FAT file system (large USB-based harddisks may come formatted with NTFS nowadays). Unless Philips provides a custom device driver with every device sold, FAT is the mandatory file system.

The situation is likely to change for the next generation of connectivity standards. Therefore it's interesting to measure different file systems and see if it influences the performance of the testcases. Currently only the Ext2 file system is measured as it is the default Linux file system.

Different Linux versions

The Linux kernel is currently at version 2.6.13. However, the use of the 2.4.x Linux series is still very common in embedded devices. Measuring different kernel versions may be interesting as newer device drivers not always result in a better performance. Changing the kernel version will require a reboot of the system, so it's rather time consuming to measure the difference in performance between kernel versions.

Different hardware platforms

[3] See the manual page of proc(5) under Linux

As described in chapter 7, with only one day available to perform the measurements, a benchmark run has been completed on the ARM architecture. Unfortunately, only SQLite with and without Libdbi compiled correctly on the platform. If there were a few more days available, also MySQL and BDB would have worked. It's valuable to see if these DBMSs would stay under the 100ms limit that is defined in section 2.3.

9.3.4 Identify cache influence

As described in section 3.4.2, several layers of cache exist in a computer. This subsection identifies two caches that may be useful to control in order to stabilize measurements further.

OS page cache

Detailed information on controlling the OS page cache is documented in [Mes03]. The author of the document also knew of a patch[4] (developed within Philips Natlab) for the Linux 2.4 kernel to control the (V)FS cache.

Query cache

DBMSs often feature a query cache. As an example of such an DBMS, MySQL is used as there is good documentation [AB05] available on the subject. From MySQL version 4.0.1 and on, SELECT queries and their results can be cached (caching is disabled by default). If an identical query is received by the server later, the results are retrieved from the query cache rather than parsing and executing the query again.

The MySQL authors claim a 13% performance loss for having an active query cache that's not being used (when no repetative queries occur). A minimal speedup of 238% is gained if a query is a cache hit.

Tests on real users would provide details about the number of repetitive queries. The number of cache hits and misses are automatically registered by MySQL. If the number is high, it could be beneficial to enable the query cache at the expense of a performance penalty on "one-time" queries.

[4] Just before this graduation project ended, a new patch has been found on: `http://adlr.info/?Cfree`.

A different method of caching which is at least supported by MySQL is the use of caching hints in the query. The hint lets the DBMS know if the results of the query should be cached or not. The difficulty is that the application should be able to determine the situations in which caching is beneficial.

9.3.5 Parallel workloads

The current benchmarking framework only runs one query on one DBMS at the time. Multiple instances of the benchmarking framework can be run in parallel but there's no synchronization mechanism to let multiple queries begin at exactly the same time. To simulate a multi-user workload, the benchmarking framework needs to support more than one connection to a DBMS simultaneously. This requires threading in the benchmarking framework. No work towards this goal has been undertaken due to time constraints and the low priority of this task. For improving scalability measurements, especially for target platform 3, it's important to implement this.

9.4 Towards a more accurate benchmarking framework

The benchmarking framework is currently able to perform very accurate measurements (below 1 ms), but it can always be made more accurate in other ways.

9.4.1 Dummy queries

The idea of dummy queries is introduced in appendix B. Several sources of distortions were identified during initial measurements. To counter the effect of caching during benchmarks, all layers of cache (described in section 3.4) should be cleared before the start of a measurement. A way to achieve this effect is executing a batch of queries specifically designed to pollute the caches by requesting all kinds of data that is not relevant to the query that will be measured right after the dummy queries. The implementation has never been realized. Due to time constraints but also because the situation seems to be resolved in chapter 6 where the benchmarks are performed on a clean system. At the time, the queries, datasets and used DBMSs in a measurement were already run in random order. The chance of having a cache hit is small.

9.5 Recommendations

Before embarking upon ambitious trails like measuring energy consumption, I'd recommend to first pick the low hanging fruit. Spending a few extra days to perform more benchmarks on the ARM architecture would be very useful. It's important to see if BDB and MySQL still perform well when the resources are limited.

Interesting results can be expected when the benchmarking framework would receive client programming language independent capabilities. What would happen to the performance when the DB API doesn't just copy results from the database to the application (which takes a long time, as is concluded in section 6.6) but works only with pointers that point directly into the data store. Maybe there exists a small, embedded DBMS that is tightly integrated with its programming language.

Bibliography

[AB05] MySQL AB. *The MySQL manual.* MySQL AB, 2005. URL: http://dev.mysql.com/doc/mysql/en/index.html.

[BvGS02] M.P. Bodlaender, J.P. van Gassel, and N.W. Schellingerhout. *PSC Issues & Opportunities, Mobile Content Management & Connectivity.* Philips Research, 2002. NL-TN 2002/824.

[BZ04] Derek J. Balling and Jeremy Zawodny. *High performance MySQL.* O'Reilly, 2004.

[EN00] R. Elmasri and S.B. Navathe. *Fundamentals of database systems.* Addison-Wesley, 2000.

[JHB02] P. Cano J. Haitsma, T. Kalker and E. Batlle. *A review of Algorithms for Audio Fingerprinting.* Proc. Int. Workshop on Multimedia Signal Processing, St. Thomas, 2002.

[Kli04] Keven E. Kline. *SQL in a nutshell, a desktop quick reference.* O'Reilly, second edition, 2004.

[Mes03] O. Mesut. *Hard Disk Drive Energy Saving Strategies for Personal Infotainment.* Philips Research, 2003. Philips Research Technical Note PR-TN-2003/00499.

[Nie94] J. Nielsen. *Usability Engineering.* Morgan Kaufmann, 1994.

[Pac03] A. Pachev. *MySQL enterprise solutions.* John Wiley and Sons, 2003.

[Wan00] Q. Wang. *Cost-Based Object Query Optimization.* Oregon Graduate Institute of Science and Technology, 2000.

[WFS05] J. Nesvadba W.F.J. Fontijn and A. Sinitsyn. *Integrating Media Management towards Ambient Intelligence.* 3rd Int. Workshop on Adaptive Multimedia Retrieval, Glasgow, UK., 2005.

Appendix A

Terms

- ARM: The company "Advanced RISC Machines Ltd." ships ARM (embedded RISC) processors and architectures.
- API: Application Programming Interface. An API is used to control an (external) application (component) in the main application.
- BDB: Berkeley DB. An open source database aimed at developers. Supported by `http://www.sleepycat.com`
- DBMS: a database management system is a system, usually automated and computerized, for the management of any collection of compatible, and ideally normalized, data. (Definition from `http://wikipedia.org`)
- Database: a database or dataset is a particular collection of related and structured data.
- MDB: the Meta-data DataBase project. Part of MMA.
- MDB-schema: the database schema used in the MDB framework. See figure D.4.
- MDB-database: a MDB-schema instantiation in MySQL.
- MDB-framework: the API and helper utilities that use MySQL and the MDB-schema for storage.
- MDB-dataset: a collection of SQL statements that relate to the MDB-database.
- MDB-subset: a subset of the MDB-schema.
- MTM: Media Transfer Management. A layer in the Media Management Architecture that handles the transfer of assets to other devices.
- MMA: Media Management Architecture. The high level architecture containing MDB, MTM and several others.
- ODBC: Open DataBase Connectivity, a standard database access method developed by the SQL Access group in 1992. The goal of ODBC is to make it possible to access any data from any application, regardless of which DBMS is handling the data.
- SQL: The Structured English Query Language is the widely used query language for relational DBMSs.
- XML: The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language for creating special-purpose markup languages.

Appendix B

Initial benchmarking framework experiments

B.1 Introduction

This is the first benchmark run in a series of three runs. The already functional components of the benchmarking framework are verified and the still dysfunctional components are identified for repairs. By benchmarking how well the DBMS candidates scale under various conditions by variation in dataset size, (in)correct behavior of the benchmarking framework is shown. The used test conditions are derived from the use-cases in section 2.8 and are more accurately defined as an instantiation of the benchmark framework in section B.2.

B.2 Benchmark parameters

Hardware platform:

At the time of performing this benchmark, the framework was not yet working[1] on an embedded platform, so only target 3 (home server) is used as hardware platform.

DBMSs:

This is the first benchmark conducted using the almost feature complete framework. At this time, tests for the SQLite and MySQL RDBMSs have been implemented. Both with and without Libdbi. Also tested is MDB. More DBMSs will be included in later benchmark runs, but not before it shows from the benchmark that the framework operates correctly.

[1]In chapter 7, an embedded platform is benchmarked.

Schema:

Two different schemas are used in the benchmark. For testing the MDB, the MDB-schema (see figure D.4) is used. This schema is used out of necessity as it is hard-coded into MDB. For the other DBMSs, a modified subset (see figure D.1) of the MDB-schema is used for the following reasons:

- The complete MDB-schema is monstrous in size and complexity. By using a subset of the schema, the benchmarks will be just as valuable but without the size and complexity. This reduction will lower the burden placed upon future developers who may want to add a new DBMS to the framework.
- There is an already existing MDB file importer that is capable of creating MDB-datasets of user-defined sizes. Without much effort, such a dataset can be made compatible with the MDB-subset.
- The project manager decided the focus is currently on investigating MDB performance.

Although the schemas differ in layout, they allow similar[2] queries to be performed.

Datasets:

The dataset sizes used in the benchmark are: 10, 20, 40, 100, 200, 400, 1000, 2000 and 4000 assets. The dimension and contents of these datasets are explained in section B.7. Bigger datasets than 4000 assets are currently impractical due to the duration of the benchmark, which is already more than 4 hours. The framework is not the cause, it handles benchmarking for such a period of time perfectly well. One DBMS under test is very inefficient, as is revealed in section B.3[3].

Dimensions:

The results of each testcase are two graphs. The two graphs show the amount of time it takes to execute the query of the testcase and to retrieve the results. One graph contains this information for MySQL and SQLite, with and without Libdbi. The other graph visualizes MDB. Because of the large time scale required on the y-axis for MDB, the graphs are separated as the results of the other RDBMSs could otherwise not be seen clearly anymore.
To get an impression of the MDB performance, MySQL with Libdbi is also shown in the second graph. Inside MDB runs the MySQL with Libdbi storage backend, so it shows the overhead of the MDB frontend. See figure **??** for the MDB architecture.

Iterations:

For each testcase, all datasets are measured only once for every DBMS. No multiple iterations due to the benchmark duration.

B.3 Testcase 1: database creation and filling

Purpose of this testcase:

Measuring the performance of the first use-case: Joe bought a brand new PIC. He wants to copy all his music from his laptop to his PIC.

The query in words: insert all asset metadata into the database.

The query in SQL: *INSERT INTO Artists VALUES (...);*
INSERT INTO Albums VALUES (...);
INSERT INTO Assets VALUES (...);

[2]As in semantically similar, using high-level queries such as "return all artists" to hide implementation details.
[3]The graphs are of low quality and unfortunately unreproducible because at the time of conducting the benchmarks, the source-code versioning system record changes in the benchmarking framework wasn't used yet.

Amount of data written in MB	Speed in KB/s
50	45174
100	43934
200	44610
300	44098
500	44027
1000	43789
2000	39885
5000	39577
10000	39866

Table B.1: Bonnie++ throughput benchmark on a WDC, 200GB harddisk with 8 MB cache.

The expected outcome of this testcase:

The INSERT performance is estimated as shown in figure B.1. It's modeled by adding database search time and harddisk write time. The balanced tree is the default data structure for indexes in most DBMSs. Searching an index takes $O(h)$ time, where h is the tree height. This component is modeled as the step function: $int(log(x)/log(2))$.

Estimating harddisk performance is difficult. Performance depends on various parameters:

- Number of platters
- Number of revolutions per minute
- Amount of cache
- Seektime
- Throughput (controller, protocol)
- File system

For a general impression, the harddisk performance on a normal desktop PC is measured. Listed in table B.1 are the results of the bonnie++ benchmark [4].
Data starts to get written at the harddisks edges where, in the same time interval, there's more magnetic medium to be written than in the middle. But when writing large amounts of data, the data gets written at a slightly lower speed when the head moves towards the middle of the platter. The harddisk write speed is considered constant for the benchmarking framework. This harddisk performance is represented in figure B.1 by: $y = x$.

[4] Bonnie++ is a benchmark suite that is aimed at performing a number of simple tests of hard drive and file system performance. Its website: `http://www.coker.com.au/bonnie++`

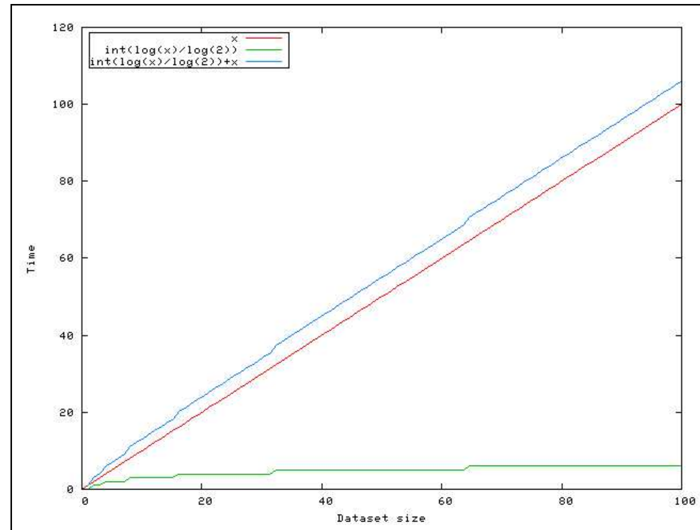

Figure B.1: Estimated INSERT performance decomposed in database and harddisk components.

The observations from the results of this testcase:

The results are visualized in figure B.2.

- The lines grow linearly overall, which is as expected. In theory, the line is slightly logarithmic because of the logarithmic tree look-up operation inside the DBMS engine. But this effect is not visible in this benchmark because of the relative small datasets. The mostly linear harddisk performance is dominant.
- SQLite with Libdbi performs better than SQLite without Libdbi. The addition of a software layer would lead one to expect it to be slower, though there is a reason for this behavior. See section B.3.2 for an in-depth analysis.

The conclusions drawn from the results of this testcase:

- The slowness of MDB is caused by the different goals it satisfies than the other DBMSs. The query language for the MDB-framework is not SQL, but a proprietary, object-oriented one. Its insert operation works by providing it with a file name. The MDB-framework then scans the multimedia file for its meta-data (e.g. ID3-tag of MP3 files or EXIF data of JPEG files.) and stores the meta-data in its storage backend that is currently MySQL. For the other DBMSs, the scanning and storing steps are separated because scanning doesn't belong to basic DBMS functionality and is therefore executed before the actual measurements. Separating these two functions within the MDB-framework is not trivial, thus the MDB-framework can't be compared to the other DBMSs in this testcase.
- The difference in time between MySQL and SQLite in figure B.2 is not a DBMS issue. Examination of the framework suggests it's also not a framework issue. It's because of the dataset. The datasets are semantically identical, but their syntax is not. See section B.3.1 for an analysis.

These conclusions invalidate any comparison drawn between MySQL, SQLite and MDB.

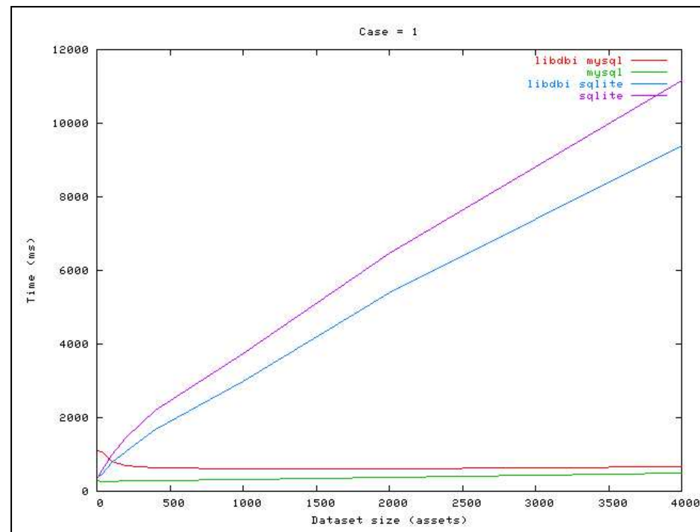

Figure B.2: Analysis 1, case 1. Total measured INSERT time per dataset in a fresh database.

Action points:

- Modify the SQL statements used in the benchmarks, in such a way that the statements adhere to the SQL standard [Kli04] and that the statements are accepted by the lowest common denominator DBMS. This ensures semantic and syntactic similarity.
- Indicate the measured points that should be included in the graph for more insight in the measured and interpolated data.
- Separate concerns by first producing a correct framework, establish the correctness of the framework by measuring DBMSs that let themselves predict better than MDB (chapter 6). Afterwards, benchmark MDB again (chapter C).

B.3.1 MDB file importer

To allow automated testing of the MDB-framework, it comes with a file importer.
The file importer takes an existing directory with multimedia files as input. A directory of MP3 files for example. Using recursion, all MP3 files are found and their ID3-tag (containing meta-data), is read and converted into SQL statements. Those SQL statements are immediately executed and results are put in the MDB-database. To use the MDB-dataset for the benchmark, it has to be extracted from the data that is in the MDB-database now.

As written in the preamble of section B.3, extracting (or *dumping* as it is called by MySQL) the MDB-dataset in a straight-forward manner, has two disadvantages:

- The extracted MDB-dataset has MySQL specific SQL commands. These dialects are not accepted by SQLite.
- The extracted MDB-dataset has extraordinary long INSERT lines (batch inserts). One line per table. This is a MySQL-only feature. It allows fast inserts in MySQL, as can be seen in figure B.2, but SQLite doesn't support this either[5].

To level the playing field, the extracted MDB-dataset is converted into more general SQL commands, understood by all RDBMSs. Every long INSERT line is split into multiple small INSERT statements.
Of course, these conversions are performed before the actual measurements.

The fairness of disallowing MySQL the speed advantage is a point of discussion. In a server environment, massive amounts of writes may happen frequently. But in the embedded environment, writes will be performed less frequently.

B.3.2 Libdbi

Libdbi [6] implements a DBMS-independent abstraction layer in C. By writing one generic set of database operations, programmers can leverage the power of multiple DBMSs and multiple simultaneous DBMS connections by using this framework.
Libdbi is used in the MDB-framework to provide abstraction from the actual DBMS.

Generally, DBMS abstraction has the following disadvantages:

- Often faster, DBMS specific SQL code can't be used.
- An abstraction layer slows down DBMS operations because of the extra layer of code.

In the particular case of SQLite and Libdbi in figure B.2, an abstraction layer seems to speed up the queries. These may be the possible causes:

- As described in figure B.3, the framework uses the SQLite C API (v3) for direct communication with SQLite. When the framework uses Libdbi for SQLite communication, the SQLite C API (v2) is used underneath. The different APIs may cause difference in overhead. In the next benchmark run, both versions of the SQLite C API will be tested. Libdbi can only use version 2 of the SQLite C API because Libdbi is old, unmaintained software.
- SQLite 2 is typeless (SQLite 3 is typed). Everything SQLite returns from a query is a string. Conversion is usually done by the caller (benchmark framework). The SQLite engine has no knowledge of types, but if the type was mentioned in the schema at table creation time, it can be always be derived from the schema afterwards.
  Libdbi is smart in a way that it derives the type from the MDB-schema. This may lead to faster conversion [7].
- If Libdbi runs as a different process, it may have a different priority. This could affect performance. Because such a process is spawned for a very short time, conventional tools (polling all processes every x seconds) to discover the priority won't work. In depth OS knowledge is needed to prove a priority.

[5] Unless using transactions.
[6] The Libdbi website: `http://libdbi.sourceforge.net/`
[7] See the Libdbi-drivers source archive: "libdbi-drivers-0.7.1/drivers/sqlite/dbd_sqlite.c"

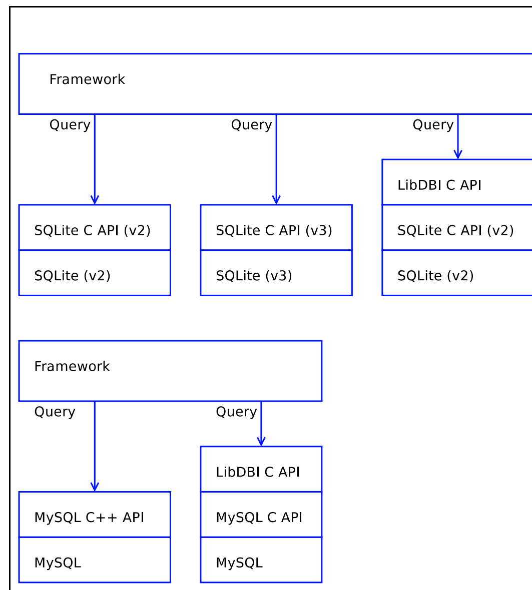

Figure B.3: SQLite, MySQL with and without Libdbi

B.4 Testcase 2: simple scan

Purpose of this testcase:

Measuring the performance of the second use-case: John wants a quick overview of all music on his PIC. He just wants a list of songs.

The query in words: return the complete details of all assets. (Indicates raw performance.)

The query in SQL: *SELECT * FROM Assets (LIMIT 100)*

The expected outcome of this testcase:

For search operations where a filter is defined (a WHERE clause in SQL) holds: The bigger the dataset, the longer it takes for the indexes to be searched and the more time the harddrive needs to retrieve requested data as data portions are physically more distant from each other.

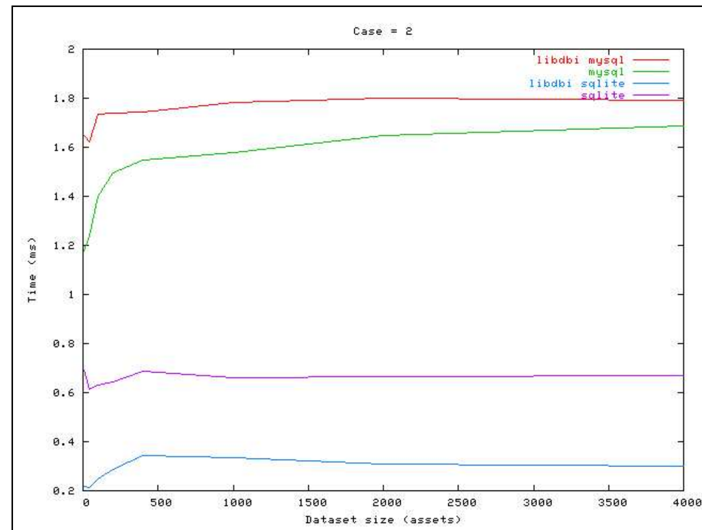

Figure B.4: Analysis 1, case 2. Total measured time to retrieve 100 records from a table. Measured per dataset.

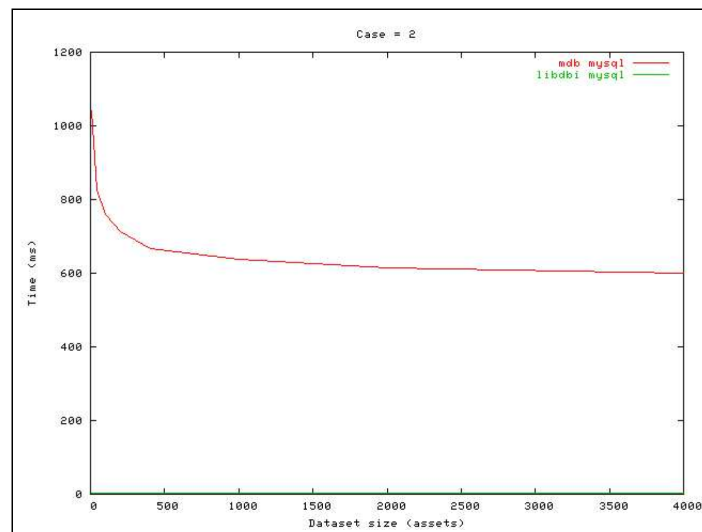

Figure B.5: Analysis 1, case 2, MDB. Total measured time to retrieve 100 records from a table. Measured per dataset.

In this testcase however, any 100 returned values from the requested table are valid. It's up to the DBMS to show that it picks the data that is clustered together the most on the harddrive to minimize the query execution time. Unlike a search operation, the index is not used in this testcase. So the performance comes down to the overhead in the query parser plus transportation time of data from harddisk to the user application.

The observations from the results of this testcase: The results are visualized in figure B.4 and B.5.

- The sudden drop of the lines around datasets of value 50 is unexpected. It translates to: insert more elements in less time. It may be normal DBMS behavior or a distortion of 0.1ms. Section B.5 contains an analysis of the possible sources and measures of distortions.
- The DBMS with Libdbi seem to perform better at bigger datasets. This may also be the result of a distortion. Measurements on bigger datasets are also required to increase insight in this issue.
- The LIMIT directive may be the cause of unexpected results. This will be investigated in the next benchmark run.

The conclusions drawn from the results of this testcase:

- The MDB-framework performance is below expectation. Upon examination, a bug revealed itself in the MDB-framework. For every returned asset, many extra, unnecessary internal queries are performed. This behavior was noticed when every query MDB sent to Libdbi was logged. The source code for logging the queries is made available in GForge [8]. Upon investigation, it seems not be just a bug but a serious design issue. A very general function inside the MDB-framework retrieves all kinds of data, regardless the initial query. The problem has been reported and acknowledged by the MDB programmers. It's out of this projects scope to act on MDB design issues.

B.5 Distortions

Finding the exact source of measurement distortions requires in-depth hardware, Linux kernel and/or operating system knowledge neither my colleagues nor I possess. This knowledge is not easily acquired. Therefore, it's beneficial to first attempt the easy solutions.
Several classes of distortion can be distinguished and are explored in the following subsections.

B.5.1 Other processes

The problem:

The benchmark is currently performed on a desktop system that performs unclear tasks in the background at random intervals. This is not a good environment to perform benchmarks in. Linux, the OS that runs on this PC, is a multi-tasking OS.
Multi-tasking [9] is the ability to run more than one program at once "simultaneously". Actually, without a certain setup of multi-processors, the computer still does one thing at a time, but the system has the ability to do time-sharing between processes, scheduling multiple programs' usage of the processor so that it appears to be running several things simultaneously.
To achieve reliable measurements, the amount of time-sharing should be minimized. Minimized, not eliminated, because some OS processes are critical.

[8] The Natlab internal source code repository
[9] Definition from: `http://wiki.linuxquestions.org/wiki/Multi-tasking`

The solution:

Take a new PC, dedicated to the benchmark. Perform a clean Linux installation. Remove all processes non-critical to the OS and the benchmark. Then perform the benchmarks.
There are no better alternatives that are as easy. The benchmark can run inside a virtual machine for example. But it will affect time measurements. Although processor cycles can be counted instead of elapsed time, this requires a new benchmark framework.

B.5.2 Caching

The problem:

Caching is a problem[10] that spreads throughout multiple layers of software and also hardware. The complete caching topic is described in-depth in section 3.4. The caching problem may be at the root of the unexpected measurements in figure B.4. All measurements are performed in order of increasing dataset size. This may explain why the second query is executed faster than the first query in some occasions. The result of the second query may contain the cached result of the first query. One level below the DBMS, at the File System level, caching may also be at work. All pages of data that were retrieved for the first query can be kept in the harddisk- or OS page cache. Even a few more blocks because of the often enabled read-ahead feature on the harddisk. Because of this feature, the second query may be even performed faster than the first.

The solution:i

Performing the measurements in non-increasing order (i.e. random) is no guarantee for elimination of caching behavior (even when assuming the size of the cache(s) is limited and reasonably small), but it's a simple start.
A general solution for cleaning the cache is performing a series of 'dummy' queries that will flood the cache so that results of earlier testcase queries are thrown out of the cache. The definition of what the 'dummy' needs to be, depends on the cache implementation of each DBMS. Restarting the DBMS also helps in some cases.

B.5.3 Other Intermittent Distortions

The problem:

Whenever the source of distortion can't be controlled (i.e. kernel events), there are general solutions to minimize the effect of the distortions.

The solution:

To iron out spikes in measurements, perform the measurements multiple times and derive the average value. Additionally, show the standard deviation.

B.6 Testcase 3: inner-join

Purpose of this testcase:

Measuring the performance of the fifth use-case: Jill wants to see a list of available albums on her PIC. After selecting an album, she wants to see what songs are in it.

The query in words: return a complete album list consisting of album-, and artist name. (Join with Albums and Artists tables. A typical view a music-browser would request.)

The query in SQL: *SELECT * FROM Assets AS ass, Albums AS alb WHERE ass.AlbumId=alb.AlbumId (LIMIT 100);*

[10] Usually, caching is positive behavior. But depending on what is to be benchmarked, it may skew the results.

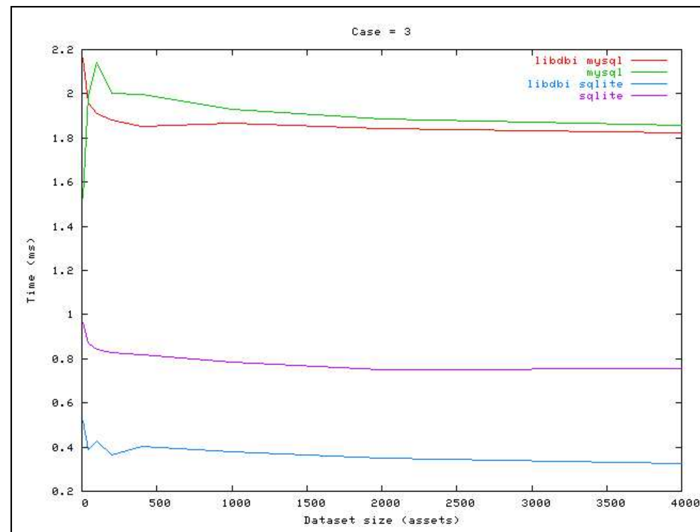

Figure B.6: Analysis 1, case 3. Total measured time to retrieve 100 records from an inner-join. Measured per dataset.

The expected outcome of this testcase:

Because of the LIMIT 100, all datasets bigger than 100 should perform likewise.

The observations from the results of this testcase:

The results are visualized in figure B.6 and B.7.

No new, meaningful conclusions can be drawn before solving the issues that were also present in the previous testcases. Further analysis of queries is gratuitous as the benchmark is currently proven unreliable.

B.7 Datasets

The MDB-subset contains only the following three tables: Assets, Albums and Artists. The fields are identical to the ones in figure D.1. The MDB-subset is used because of its simplicity, practical real-world value, and join ability.

The dataset size of 10 refers to 10 records in the asset table. Those 10 assets may come from the same album and artist, but may also differ. In total, there's a total number of records in the three table between 12..30. Unfortunately, this is not very consistent. In the new dataset generator, this ratio is always kept at 10:1:1.

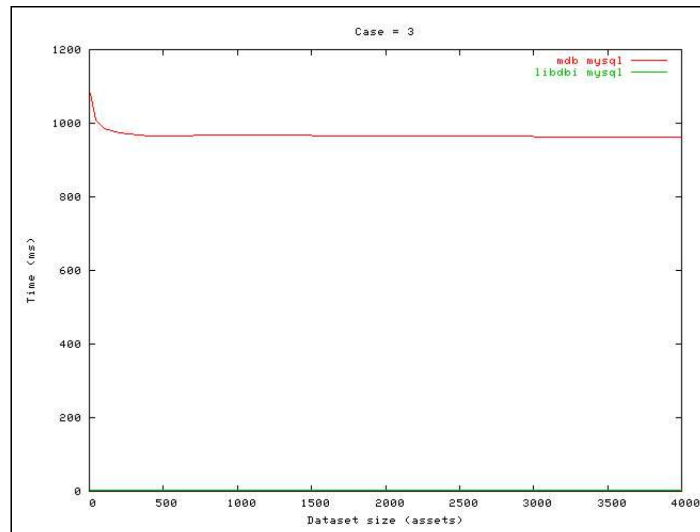

Figure B.7: Analysis 1, case 3, MDB. Total measured time to retrieve 100 records from an inner-join. Measured per dataset.

The MDB-subset is correct, but the collection of data generated for it could be improved, as explained below.
As described in section B.3.1, the dataset is generated from meta-data inside MP3 files and the file name itself. To generate a dataset of 10000 assets, for example, 10000 MP3 files are read by the file importer.
Using symbolic links (symlinks), 10000 files can be created while occupying only the space of, say, 100 MP3 files. This is a welcome feature because of the otherwise enormous collection of test MP3 files that ought to be acquired and stored. As shown in figure B.8, a couple of new symbolic links are created, linking to a real MP3 file.
File names the file importer will encounter are unique. That results in 10000 unique assets in the Assets table. But the Albums and Artists table won't grow proportionally because the meta-data, unlike the file names, won't change using symbolic links.
The size of the Assets table is proportional to the amount of MP3 files (real files and symbolic links) scanned by the file importer, but for every symbolic link scanned, the size of Albums and Artists tables will remain equal.
A new design to supersede the file importer is presented in section 6.2.

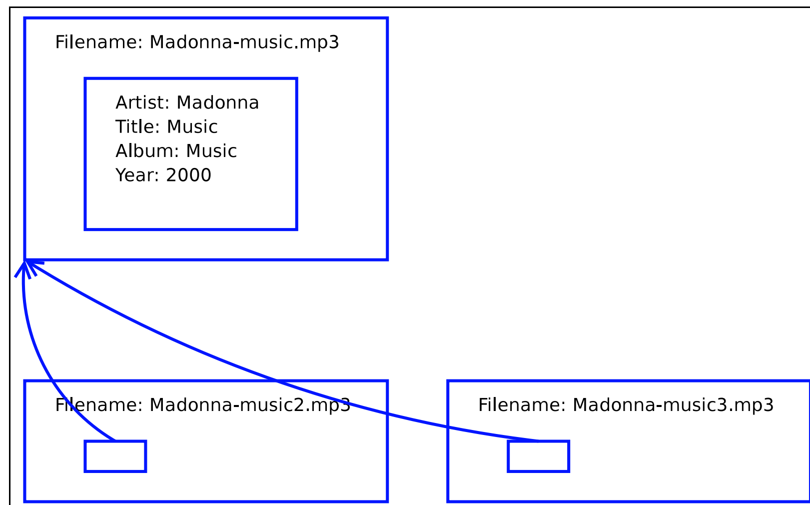

Figure B.8: Symbolic links in Linux

Appendix C

MDB Analysis

C.1 Introduction

This appendix shows the solution to the question:

In which layers of figure C.1 are the biggest speed bottlenecks?

It's the last unanswered question from the requirements in section 2.5.

The MDB C++ API layer is a theoretical concept. In practice, the MDB C++ API and Object-Schema mapping layers are interwoven in the source code. These layers are measured as one entity. Also the Libdbi, MySQL and OS layers are treated as one entity. Otherwise, profiling each of these layers separately would be a time-wise expensive exercise as the Libdbi layer has to be modified to record timings.
By looking at the difference of MySQL with and without Libdbi benchmark results from chapter 6, one can conclude that Libdbi's influence is limited on at least target platform 3.

C.1.1 Measurements

The benchmarking framework, hardware and testcases of chapter 6 have been used to benchmark MDB. An MDB module has been developed and was added to the framework to enable benchmarks for this DBMS.

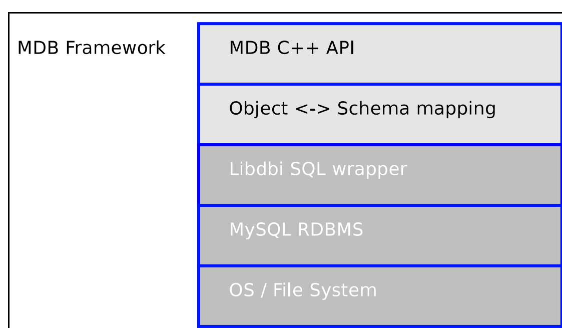

Figure C.1: The MDB framework architecture

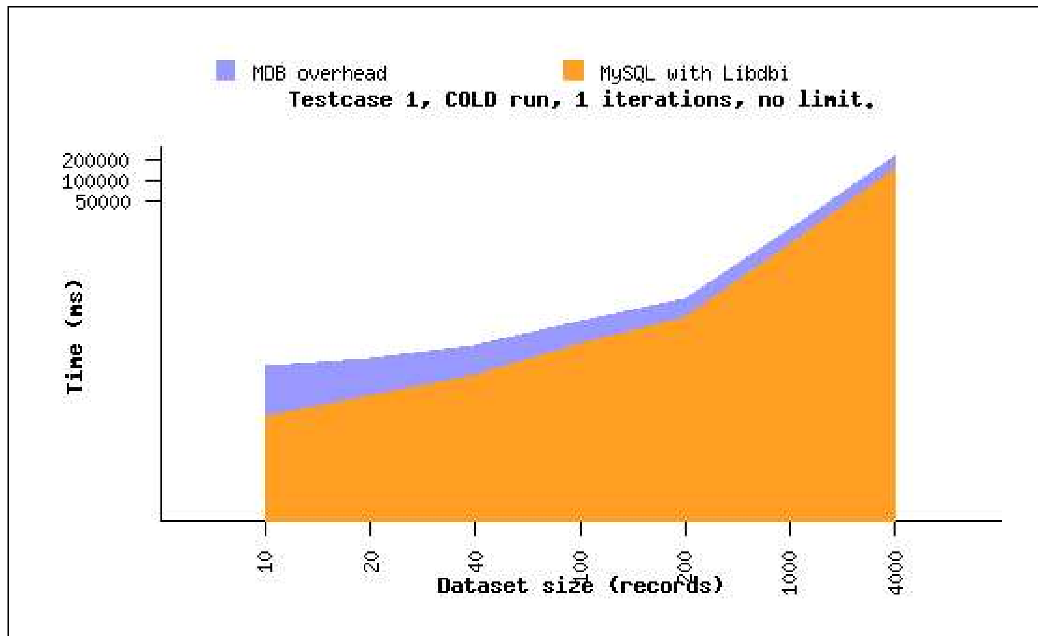

Figure C.2: Testcase 1. Total measured INSERT time per dataset in a fresh database. Measured per dataset.

Each testcase is performed with small datasets as MDB doesn't scale much further that 10000 assets. The orange area in each graph represents the time spent by the DBMS (MDB is inactive). The blue area that is added onto the orange area represents the time spent by internal MDB framework operations (DBMS is inactive). Both areas added together are the total time to complete an MDB query.

C.2 Conclusion

The time spent by the MDB framework itself is negligible, compared to the time spent by the DBMS. Except for testcase 1, where the MDB framework performs file imports that are I/O bounded.

At first sight, it looks like the DBMS is slow. This is not the case. The MDB framework is performing many SQL queries to the DBMS for one MDB query performed by the benchmarking framework. This is proven by adding a printout every time the MDB framework performs a query to the DBMS. For every executed test, MDB typically performs SQL queries 4 times the amount of assets that are in the results. For every asset that matches the MDB query, all related properties are aggregated in order to return complete asset objects to the benchmarking framework. The code to see these printouts is integrated in the main MDB branch[1] and can be activated using a compiler switch.

To improve the MDB framework, the amount of internal queries in the MDB framework should be reduced. It's not needed to receive complete objects every time. The problem may be solved by specifying what information is really needed in the returned asset objects so that not every property is retrieved by default.

[1]Only accessible for Philips employees on the intranet

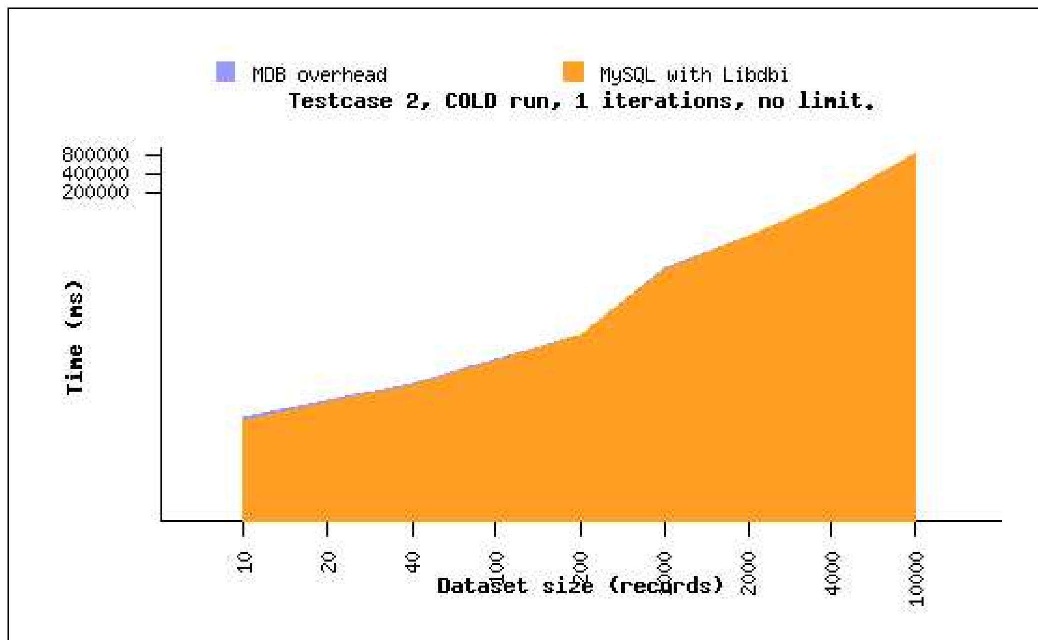

Figure C.3: Testcase 2. Total time to retrieve all records from the database. Measured per dataset.

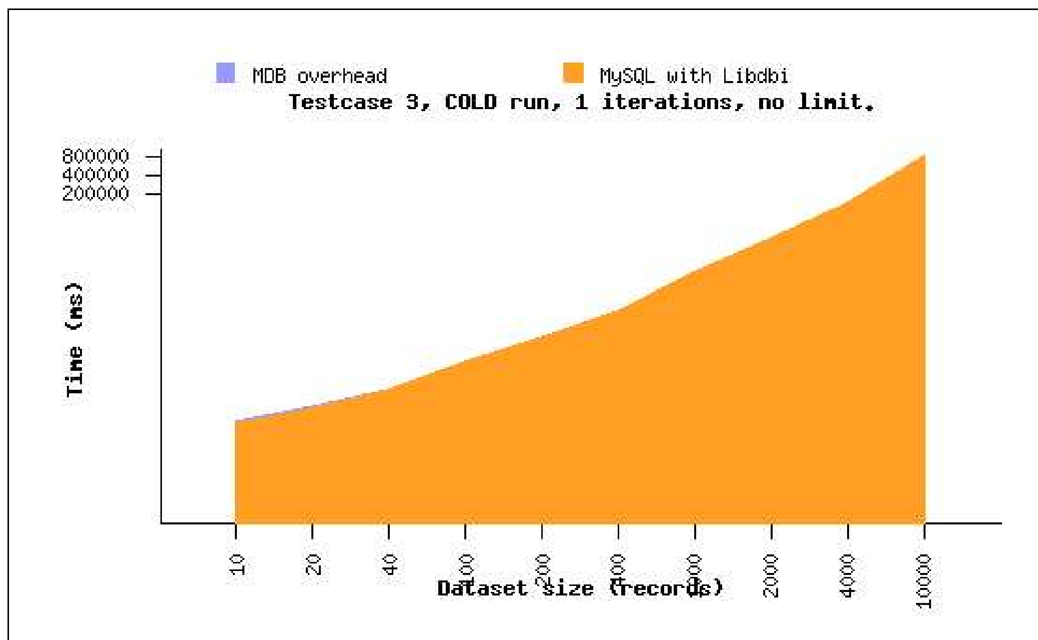

Figure C.4: Testcase 3. Total time to search for records in a database. Measured per dataset.

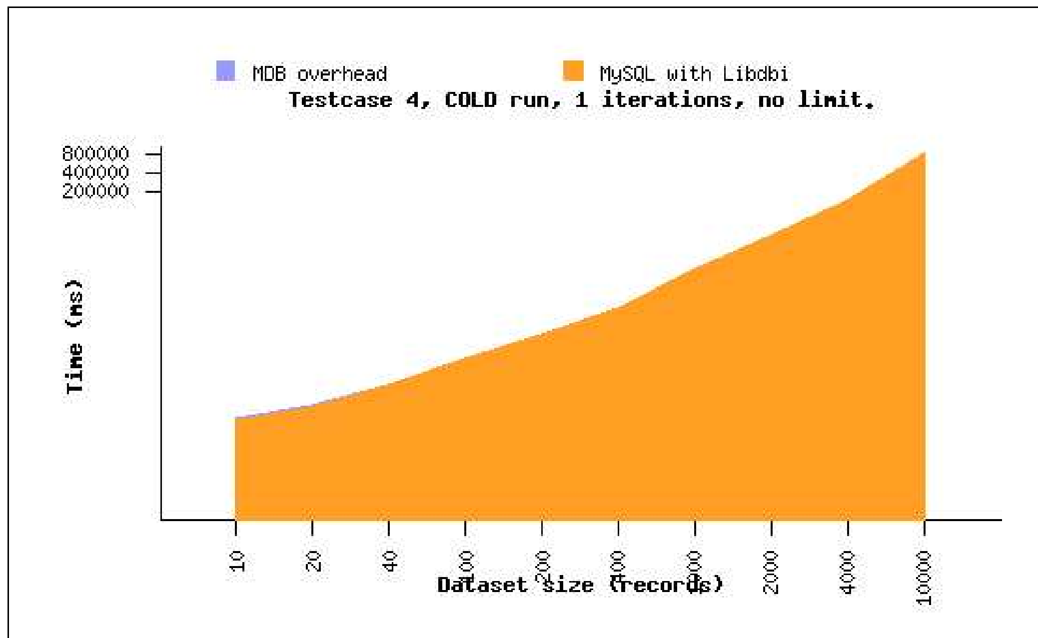

Figure C.5: Testcase 4. Total time to search for a non-existing record in the database. Measured per dataset.

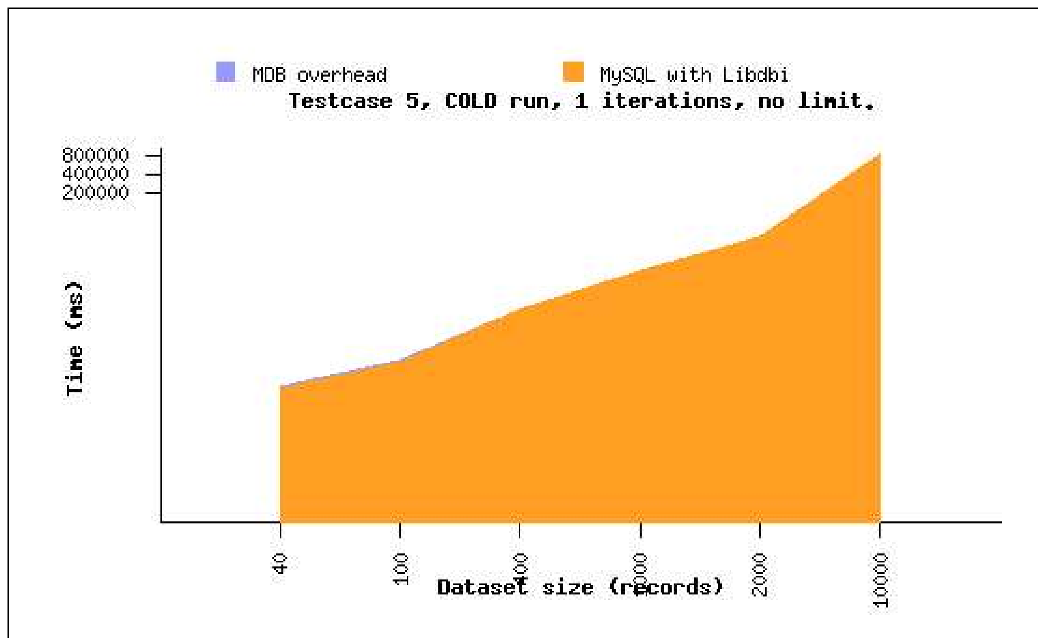

Figure C.6: Testcase 5. Total time to retrieve all records from a join. Measured per dataset.

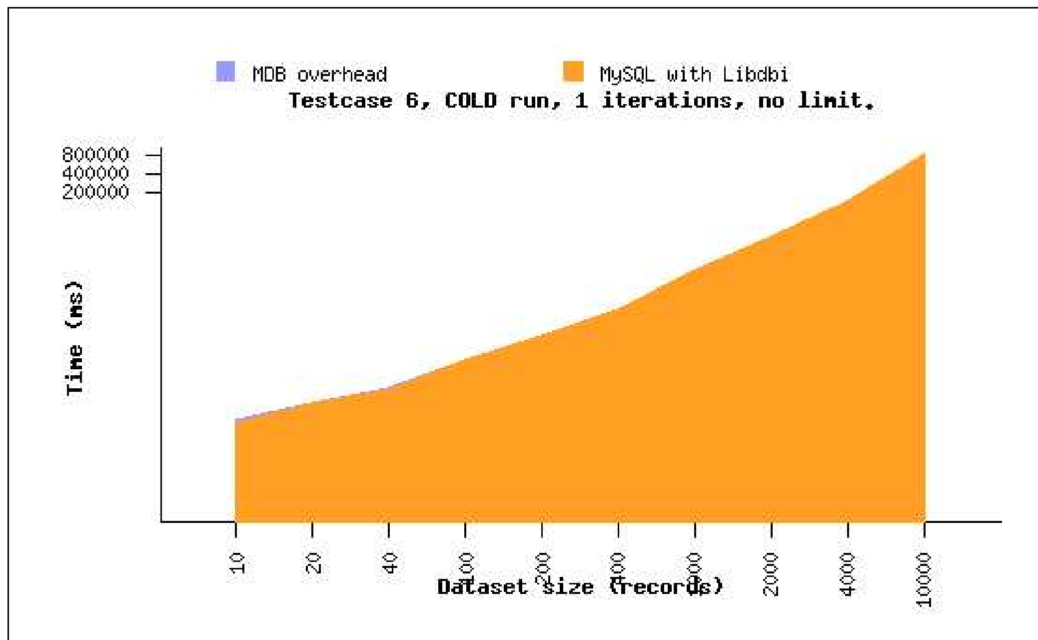

Figure C.7: Testcase 6. Total time to retrieve all records from a join with a filter. Measured per dataset.

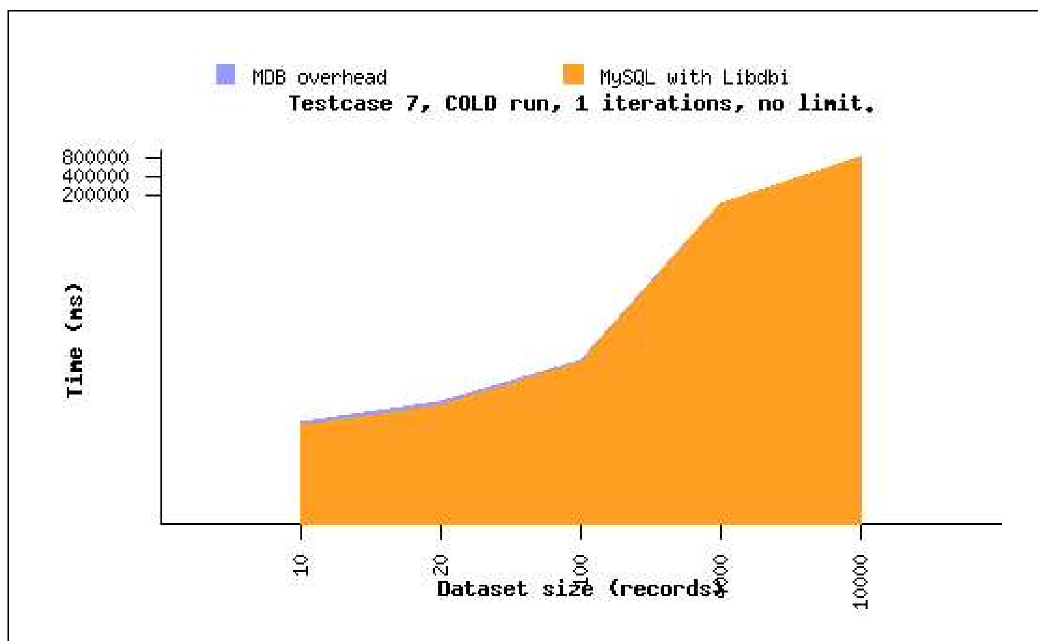

Figure C.8: Testcase 7. Total time to retrieve all records, sorted alphabetically. Measured per dataset.

Appendix D

MDB API and schemas

For an impression of the complexity of the target platforms. Their database schemas are presented in this appendix.

Figure D.1 represents the simplified schema that is used during the benchmarks.

D.1 Target platform 1, Mobile audio player

Used APIs in MDB: Asset, Relationship.
DBMS schema: see figure D.2

D.2 Target platform 2, Mobile audio/video player

Used APIs in MDB: Asset, Relationship, Person, Playlist.
DBMS schema: see figure D.3

D.3 Target platform 3, Home server

Used APIs in MDB: All
DBMS schema: see figure D.4

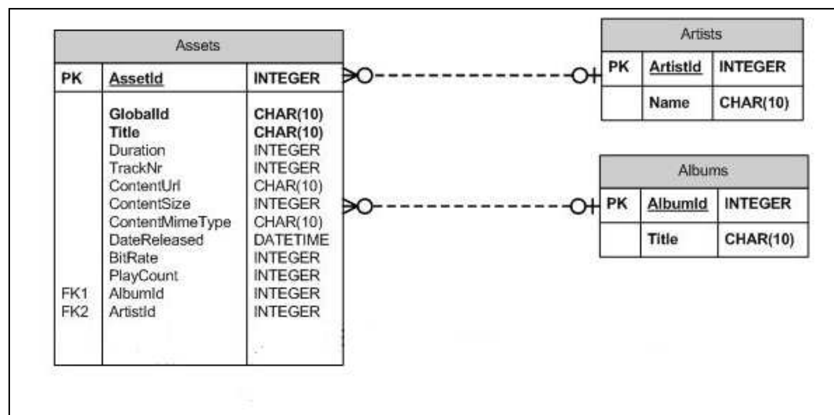

Figure D.1: MDB-subset with indexes on primary keys, Artists.Name and Albums.Title

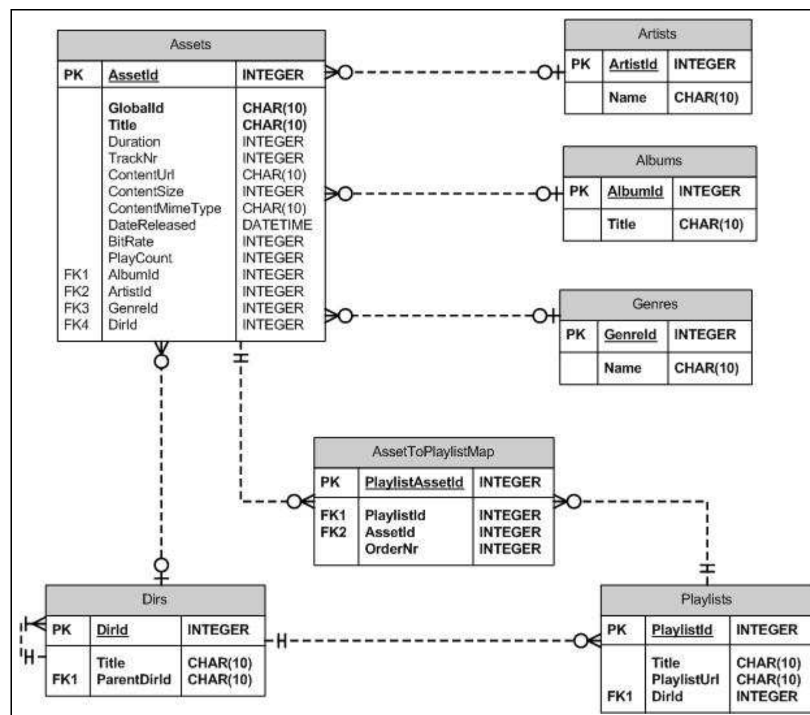

Figure D.2: Target 1 schema

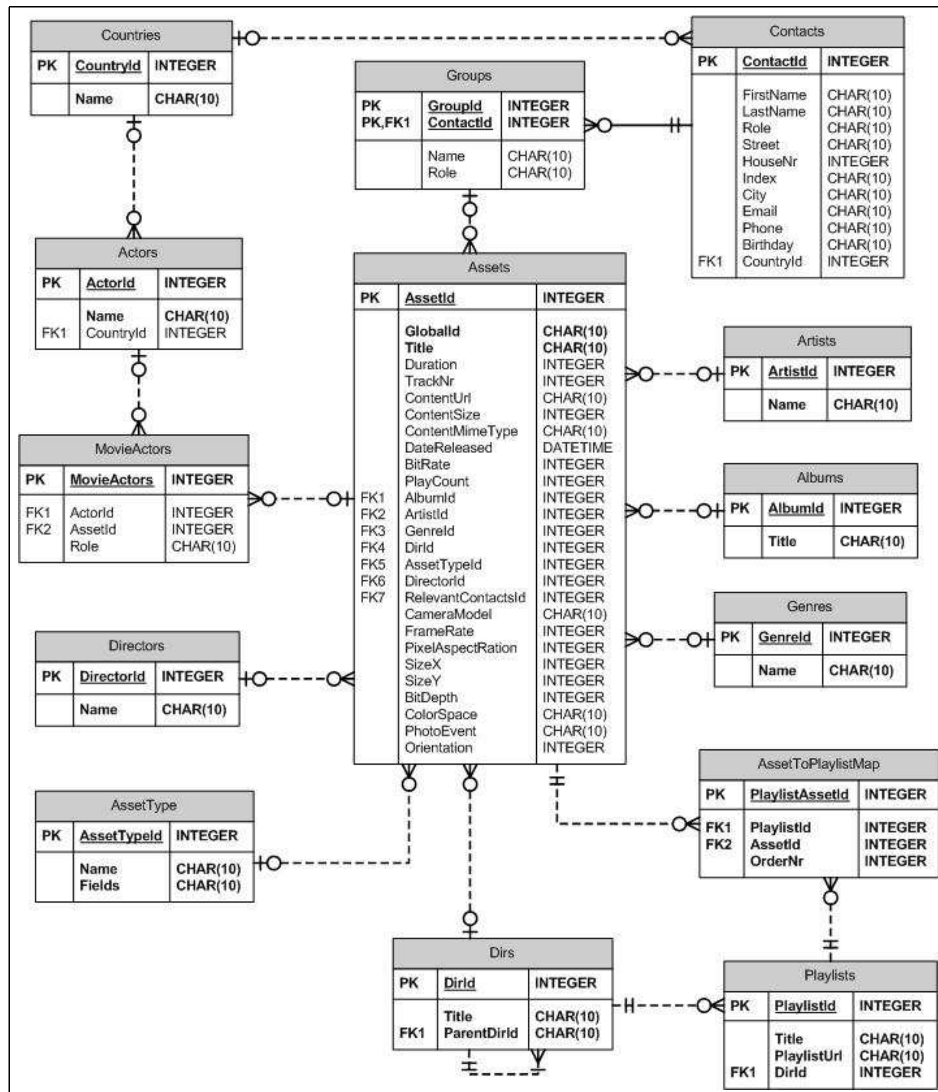

Figure D.3: Target 2 schema

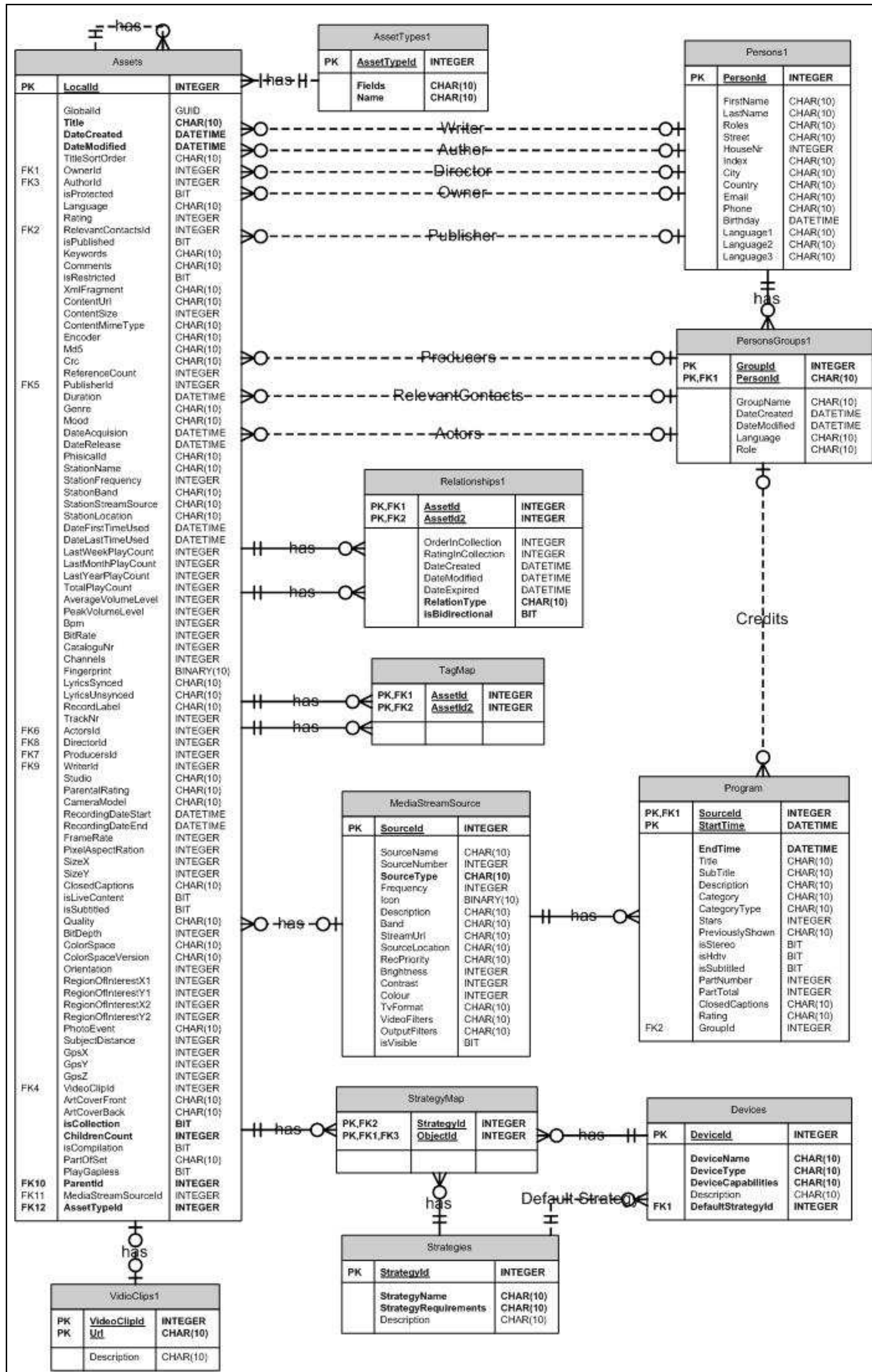

Figure D.4: Target 3 schema

Appendix E

Original project plan

E.1 Introduction

This original project plan was created during the startup period of the graduation project. The revised project plan in appendix F was made a few months after the startup period to indicate the changed course.

This graduation project started on 2004-09-01 and will end on 2005-05-30.

I carried out my graduation project within the Storage Systems & Applications group at Philips Research Eindhoven. During this time I received guidance from Stephen Cumpson at Philips and Ad Aerts at the TU/e.

E.2 Problem statement

Philips develops a new audio, video and photo jukebox for consumers. It's a portable device to organize and play your favorite media. Managing meta-data is an important issue in the project. A framework with a simple database for meta-data storage is in place. Optimizing meta-data storage and retrieval for speed and power efficiency (among other requirements) requires in depth research and is therefore made a separate project.
The software using this meta-data communicates in assets (meta-data objects) to an underlying storage layer. This layer now consists of a wrapper and a simple relational database and is believed to be a bottleneck.
Exploring the advantages of fundamental changes in the storage layer like using Object Oriented and XML databases or performing clever caching is what this thesis is all about.
The route to storage layer optimization with all its traps and choices made along the way is documented in this thesis.

E.3 Desired results

Questions answered:

- **What storage/caching system should be used?**
  - Choose a database implementation that is compliant with the requirements (which will be defined first)
  - **What are the pro/cons of RDBMSs, O/RMSs, OODBMSs, XMLDBMSs and trees?**
    Find out the database type that has the most potential.
    - Perform benchmarks:
      - Define a benchmark plan: how to test (scheme), what to test (queries), what to measure (real/system time, memory usage, ...)
      - Divide the work. (Let CWI benchmark the RDBMS, let me benchmark the others)
  - **What trade-off between speed/memory usage is optimal?**
    - How to choose what meta-data to store/cache?
      - Dynamic index layer
    - What is the minimal required cache-space to operate efficiently?

Deliverables:

- Thesis
- Database benchmark
- Prototype with database implementation

E.4 Time table

sep-oct2004: Reading
oct-nov2004: Reading, experimenting
nov-dec2004: Finalize project plan, benchmark definition draft (deadline 5 dec)
dec-jan2005: Finalize benchmark definition (deadline 1 jan), perform benchmarks, find out pro/cons of databases
jan-feb2005: Perform benchmarks, find out pro/cons of databases
feb-mar2005: Evaluate feasibility dynamic index layer, other tricks
mar-apr2005: Either continue optimizing surrounding layers / or add tricks to an existing database
apr-may2005: Implement complete prototype, write documentation
may-jun2005: Finish up

E.5 Risk evaluation

Time shortage, illness, unavailability
probability: medium
impact: high
prevention: A margin of 1 month to finish things that have been delayed.
correction: Adjust project planning first. If that isn't sufficient, drop low priority requirements.

Lack of expertise
probability: medium
impact: high
prevention: Buy and read books. Consult colleagues and Google.
correction: Discuss with colleagues from other Philips departments and the TU/e. Adjust requirements.

Project cancellation
probability: medium
impact: high
prevention: Duplicate and store available project products and knowledge. Don't totally depend on Philips availability.
correction: Work from university/home. Find someone to take over the guidance.

Changing requirements
probability: medium
impact: high
prevention: Define requirements up-front in a clear document.
correction: Discuss the need of the change. Check available time, correct planning.

Documents or products are lost
probability: medium
impact: medium
prevention: Backup the documents and software.
correction: Restore from backup.

Appendix F

Revised project plan

F.1 Introduction

This revised project plan was created after the startup period of the graduation project. It indicates the changes in direction of the research that is being and going to be performed during the remaining months. See appendix E for the original project plan.

This is the project plan of my graduation project carried out within the Storage Systems & Applications group at Philips Research Eindhoven.

This project started on 2004-09 and will end on approximately 2005-09.

During this time I received guidance from Ad Aerts at the TU/e, Alexander Sinitsyn and Joep van Gassel at Philips. Stephen Cumpson, my first mentor at Philips left to another department.

F.2 Problem statement

Philips is developing a new class of portable audio, video and photo devices for consumers within the PIC[1] [BvGS02] project. An important part of this project is the management of the meta-data about the multimedia. Therefore, Philips Research developed a framework for meta-data storage, modification and retrieval, called MDB[2] [WFS05].

A user application (e.g. the jukebox GUI) uses MDB to handle meta-data. For example: whenever the user wants to know what media created by "Madonna" is available on the device, MDB is invoked to find an answer to the question (the query). The results of this query are returned to the user application in the form of meta-data objects, called assets. These assets are currently stored using the MySQL RDBMS[3]. A DBMS provides operations to store, modify, search and retrieve data in a standardized and reliable way.

[1] Personal Infotainment Companion
[2] Multimedia DataBase
[3] Relational DataBase Management System

The earlier mentioned jukebox is an embedded device. This fact places constraints on the available processor speed, memory usage and energy consumption of the software. Researchers within Philips think that MySQL is not an optimal solution in the embedded environment. Especially an embedded environment needs to strike a good balance between constraints such as speed, energy usage and memory usage. In contrast, a modern PC always has enough energy and memory available for the tasks that are performed in an embedded environment. Although MySQL the RDBMS currently in use, MDB has a built in RDBMS abstraction layer that (ideally) allows replacing MySQL with another RDBMS without changing MDB. Therefore, Philips made the task of exploring the advantages and disadvantages of other ways to store the meta-data in an embedded environment, a graduation project. This report describes the process of finding the optimal solution(s).

F.3 Desired results

The following questions follow from the project description in section F.2:

- **What storage implementation should be used?**
  - Write up requirements to define what is expected from the storage implementation.
  - What are the theoretical pros and cons of plain text files, trees, RDBMSs, O/RMSs, OODBMSs and XMLDBMSs?
  - How well do the practical implementations perform? How to get this information?

Deliverables:

- Thesis
- Database benchmark software and results

F.4 New time table

1 sep - 1 oct 2004: Reading
1 oct - 1 nov 2004: Reading, experimenting
1 nov - 1 dec 2004: Finalize project plan, benchmark definition draft (deadline 5 dec)
1 dec - 1 jan 2005: Finalize benchmark definition (deadline 1 jan), perform benchmarks, find out pro/cons of databases
1 jan - 1 feb 2005: Perform benchmarks, find out pro/cons of databases
1 feb - 1 mar 2005: Create and evaluate other databases, like the b-tree
1 mar - 1 apr 2005: More b-tree, OODBMS and XML databases
1 apr - 1 may 2005: Implement complete prototype, write documentation
1 may - 1 jun 2005: Finish up

The time table has changed because it was only after a few months we had a better overview on the project and were able to better define what had to be done.

F.5 Actual, historic time table

sep - dec 2004: Reading
jan - jun 2005: Designing and implementing the framework simultaneously
jul - sep 2005: Writing thesis. Conducting additional measurements

Less than one month was planned for going from a benchmark definition to doing the actual measurements. A benchmarking framework was never considered at the time of creating the time table.

F.6 Risk evaluation

Time shortage, illness, unavailability
Probability: medium
Impact: high
Prevention: A margin of 1 month to finish things that have been delayed.
Correction: Adjust project planning first. If that isn't sufficient, drop low priority requirements.
In hindsight: I only had 2 days of illness. There were no other problem.

Lack of expertise
probability: medium
impact: high
prevention: Buy and read books. Consult colleagues and Google.
correction: Discuss with colleagues from other Philips departments and the TU/e. Adjust requirements.
In hindsight: My C++ knowledge was lacking and it turned out that the framework implementation in C++ became a substantial part of my graduation project. It delayed the project at least one month.

Project cancellation
probability: medium
impact: high
prevention: Duplicate and store available project products and knowledge. Don't totally depend on Philips availability.
correction: Work from university/home. Find someone to take over the guidance.
In hindsight: The project was not canceled, but my original mentor left the project and has been replaced.

Changing requirements
probability: medium
impact: high
prevention: Define requirements up-front in a clear document.
correction: Discuss the need of the change. Check available time, correct planning.
In hindsight: The requirements changed were

Documents ore products are lost
probability: medium
impact: medium
prevention: Backup the documents and software.
correction: Restore from backup.
In hindsight: No problems encountered. The thesis and software was stored in my personal Concurrent Versions System and also in Philips' CVS. Including two workstations to work on, this resulted in a total of 4 duplications.

Appendix G

Logbook

Logging started in December 2004 as the idea of logging was brought up at that time.

Week of 19..25dec
Prepare and give a presentation on the state of my research for Philips and CWI people. Look for more ways to profile/measure databases (valgrind for memory usage for example)

Week of 2..8 jan
Talked to external parties for evaluation versions of power consumption benchmarking tools. They're based on a theoretical model of hardware. Very expensive. Began implementing MySQL tests. Talked to my new company mentor.

Week of 9..15 jan
Learned a bit about C++ for the MySQL C++ API. Needed for the testing framework.

Week of 16..22 jan
Coded the benchmark control classes, talked about the architecture of the benchmarking framework. Asked lots of c++ questions at the annoyance of colleagues ;-).

Week of 23..29 jan
Improved the benchmarking framework, appended little pieces to the benchmark part of the document. 2 days of illness.

Week of 30..5 feb
Generated plots of executed query time for MySQL and SQLite 3. Created a script that automates the testing and generates plots. Added Libdbi to the testing framework. Realized that executing just the query is not enough, fetching results is also part of the measurements.

Week of 6..12 feb
Celebrated carnival. Learned more about pointer (ab)use in C++. Being haunted by a segmentation fault. 4 Days later, this seemed to be a very tiny semantic mistake, aaargh! Also Libdbi needed SQLite 2 instead of the previously used 3. There's no driver for version 3. Appended all this to the framework.

Week of 13..19 feb
Implemented testcase parameters to select a testcase. Created a python script to calculate max, min and standard deviation from a series of numbers. Performed (dis)connect benchmarks MySQL versus SQLite.

Week of 20..26 feb
Packed all random scraps of paper around the office in a digital mindmap. My desk is cleaner now, and so is my head ;-). Added testcases 2..4 to the framework. The framework now runs all test after invocating one script (called overseer) directing all tools. The resulting gnuplots are ready instantly for document inclusion! Finished up the benchmark part of the document. Began a little bit of B-Tree work.

Week of 27..5 mrt
More B-Tree work. Implement a tree using Berkeley DB. Attended a presentation about XML and the XQuery.

Week of 6..12 mrt
More implementation of BDB. Following the BDB tutorial more closely now.

Week of 13..19 mrt
Added BDB to the framework. A simple query works.

Week of 20..26 mrt
Working with Alexander on better framework portability. Fix Libdbi problems that surfaced this week by running it on an older Linux distribution.

Week of 27..2 mrt
Working with Alexander on better framework portability. Better library version number tracking. Fixed bugs found by running framework on another computer. Tried adding more features to BDB (using secondary tables) requires lots of work. Removed hard paths that were used in the framework (this is a no-no for portability)!

Week of 3..9 apr
Working with Alexander on better framework portability. Added the ability to benchmark with different datasets.

Week of 10..16 apr
Done extensive benchmarking and benchmarking framework additions (added more parameters to allow more control). Made a dataset generator to generate test records. Included the preliminary results in a presentation Alexander gave to customers. Benchmarks do not only include MySQL, SQLite, Libdbi but also MDB now. For this, we've added schema variation capabilities and converted the original MDB schema to SQLite-compatible CREATE commands. So we can now measure 2 schemas, 6 queries, 5 databases, and a range of datasets (currently 6). Benchmarking these takes a few hours to run now...

Week of 17..23 apr
Ran and improved the benchmark. General code cleanup and polish work on the framework with Alexander.

Week of 24..30 apr
Attended SCRUM presentation about project management, very interesting. Applied it to my daily routine, works as advertised! I gained better and more results this week. Added a switch to the benchmark framework compilation script. The framework can now be compiled with and without MDB. Needed for CWI to use the latest framework without MDB. Improved the BDB tests somewhat. It also runs test 2 successfully (query 1). At least 6 times faster than the B-Tree. Transactions will help. Read about hash tables. Searching key-value pairs may perform better using hash tables instead of a B-Tree.

Week of 1..7 may
Only 3 working days due to liberation day.

Week of 8..14 may
Studied the BDB tutorial and implemented the basic bits. Filling a database with a few key-value pairs and reading them again. Later implemented simple scan and simple filter in BDB.

Week of 15..21 may
Made a new plan for the coming 2 months to finish with a good enough result for both the framework and the thesis.

Week of 22..28 may
BDB framework integration. Remove hard paths from code, place debugging output between ifdefs. Document how joining works for the thesis. Fix a join segfault. Redirect normal messages to STDOUT, errors to STDERR. This way, invalid measurements are detected easier by the caller. Imported huge datasets in BDB, this needs some optimization because it's very slow.

Week of 29..4 jun
Discuss what to benchmark and analyze in the final benchmarking run. Made a strict separation between MDB goals and HDD60 goals. Ordered a secondary workstation to perform clean benchmarks on. More cleanups. Now errors can be logged to a separate file to get a quick overview. In debug mode, the output is more verbose now. A dataset generator has been created. Ploticus is now used to generate graphs. It's better than gnuplot. Measured points are now included. Updated document with architecture of the framework.

Week of 5..11 jun
Installed new test workstation with a minimal Linux installation. Measurements are really stable this time!
Continuation of last weeks work, polishing.

Week of 12..18 jun
Some document writing in the Analysis 1 chapter. Dug into Libdbi issues (segfaults), learned to work better with the DDD debugger.

Week of 19..25 jun
More writing on the analysis 1 chapter. Found out about linear harddisk performance.

Week of 26..2 jul
Received feedback on analysis 1 chapter. More writing of analysis 1 chapter. Started at analysis 2 chapter. Added record limit parameter to the framework (0=no limit, x=LIMIT x (x¿0)). Added log. scales to graphs. This removes clutter around nearby datasets. Graphs are now generated with both linear scales and logarithmic scales. Log is confusing the mind. Timers are now more accurately measuring the queries. Added SQLite2 to the framework (already had SQLite3 and SQLite 2 with Libdbi). Not only during (you won't really look at them), but now also after a test run, any errors will be reported. Chapter benchmark analysis 1 finalized. More work on analysis 2 and framework chapter.

Week of 3..9 jul
Now using the Philips CVS as source repository for framework source code. Using my personal SVN source repository for the thesis. People complained about binaries (and eps) put into it. Now I can also work on the thesis at home. Their infrastructure really drives me crazy! Proxies, firewalls, Lotus notes. Yay for secure tunnels. Standard deviation is now included in graphs. New overseer.sh parameter: cold or warm run. New massbench.sh: runs benchmarks with and without limit, cold and warm. Standard deviation over x iterations. Benchmark framework described better in thesis. Incorporated Alexanders thesis review proposed changes.

Week of 10..16 jul
More document writing. Received documents reviews by supervisors. Got positive results back of the 2 last exams I've been working on in the evenings, will never try that again!

Week of 17..23 jul
More document writing.

Week of 24..30 jul
Vacation. No work this week.

Week of 31..6 aug
Vacation. No work this week.

Week of 7..13 aug
Vacation. No work this week.

Week of 14..20 aug
Finalize measurements from the second run, analyze and describe them. Successfully demonstrated the framework to Peter Boncz, and an AIO from CWI. Prepared thesis for reviews.

Week of 21..27 aug
Added more to analysis 2 and concepts chapter. Reran a measurement from analysis 2.

Week of 28..3 aug
Ran last measurements for analysis 3.
Add analysis 3 to thesis.

Week of 4..10 sep
More feedback from supervisors. More writing. Thought I'd be finished by now. No!

Week of 11..17 sep
Even more writing, finishing up loose ends. More proofreading in the sun. CWI did some benchmarks using my framework. Just when things got interesting, I'm done.

Week of 18..24 sep
Writing, writing, proofreading, correcting.

Week of 25..1 oct
Finished the thesis. Handed in final version at the student administration for duplication.

Week of 2..8 oct
Practiced my final presentation in the PIC meeting at Philips.

Week of 9..10 oct
I'll be performing my final presentation! The next day, I'll be standing at the crossroads wondering which way to go...