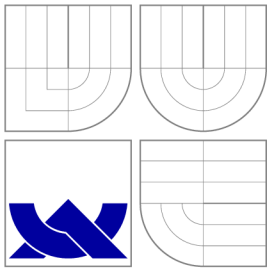
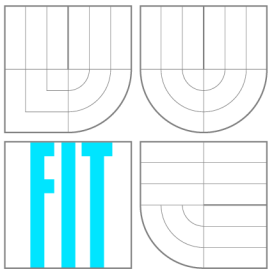


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁVRH NOVÉ RPM DATABÁZE

DESIGN OF NEW RPM DATABASE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN ZELENÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Dr. Ing. PETR PERINGER

BRNO 2010

Abstrakt

Systémy správy balíčků tvoří velmi efektivní a pohodlné řešení pro instalaci, údržbu a mazání software v počítači. Jejich základní koncept spočívá v uchovávání informací o instalovaném softwaru na jednom místě a jejich správě dodanými nástroji. RPM databáze je právě tím místem, kde jsou informace uchované. Současné řešení je zastaralé a potřebuje vylepšit. Tato práce analyzuje současné řešení, jeho možné alternativy a na základě této analýzy navrhuje nový design databáze. Součástí je také ukázka implementace rozhraní nad touto databází a porovnání se starým řešením.

Klíčová slova

RPM, rpmdm, balíčkovací systém, databáze, MySQL, SQL

Abstract

Package management systems are very effective and comfortable solution of installing, maintaining and erasing software from computer. Their basic idea is that all information about installed software is kept in one place and is managed by common utilities. RPM database is such place, where information are being kept. Currently used solution is obsolete and needs some improvements. This thesis analyzes current solution, possible alternative options and based on this a new database design is proposed. A prototype implementation is also included and it is compared with current solution.

Keywords

RPM, rpmdm, package manager, package managing system, database, MySQL, SQL

Citace

Jan Zelený: Design of new RPM database, diplomová práce, Brno, FIT VUT v Brně, 2010

Design of new RPM database

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Dr. Ing. Petra Peringera. Další informace mi poskytl Ing. Jindřich Nový Ph.D. z firmy Red Hat Czech s.r.o.

.....
Jan Zelený
May 24, 2010

Poděkování

Rád bych tímto poděkoval firmě Red Hat Czech s.r.o. za poskytnutou podporu.

Contents

Chapter 1

Introduction

Package management systems (further referred to as PM systems) are very effective and comfortable solution for installing, maintaining and erasing software from computer. They are mostly known for their usage in various Linux distributions. Although people got familiar with them mostly in GNU/Linux, these systems had been here before the time of Linux, in different Unix compatible systems. Although they looked differently, the basic principle remained the same. This basic idea and goal is that *all information about installed software is kept on one place and is managed by common utilities.*

As it results from this definition, PM systems are composed of 2 basic parts. The first part is package metadata storage, which can be represented just by directory where simple text files are stored. On the other hand, it can be represented by a complex database structure with operational API implemented on the top of it. The second part of PM systems is a set of tools which implement desired operations above the metadata database as same as various constraints defined by design of concrete PM system (e.g. dependency checking, checksum verifying, ...).

RPM is one of the oldest package managers used in Linux distributions. It is used by many vendors, the most famous distributions using RPM are Red Hat Enterprise Linux, SUSE Enterprise Linux, Fedora, Mandriva and OpenSUSE. Nowadays it is much more complex than it used to be when it started. When using "RPM" now, users mostly have the whole system working with RPM packages in mind. This system will be described at the beginning of chapter 2, which describes current RPM solution. It starts from its complete architecture and in the rest of the chapter it focuses on its particular parts. There is even a package structure described in section 2.3, because database structure is derived from it.

The goal of this thesis is to design a new RPM database. Implementation is only secondary as a proof of concept. The main reason for new design is not that current solution is bad, but rather the design is obsolete from the perspective of modern software design methodology. That leads to very complex work with the database and as a result, several flaws are in the code working with rpmdb API. They are described in section 6.1. New design should offer more straightforward work with rpmdb. This will eventually lead to simplification of the entire RPM allowing more features to be implemented easily. When designing a new database, an eye should be kept on current solution and its shortcomings. Chapter 3 does exactly that. It analyzes current rpmdb and its features in order to determine weakest links in current database design.

Current solution is based on Berkeley DB. Even though it isn't a bad solution, other and possibly better solutions exist. Chapter 4 analyzes some of these alternatives and tries to determine if any of them can be useful for new design. Section 4.1 focuses on other PM

systems. There is a number of them which use very interesting concepts. These concepts are then applied in new design. That is based on relational data model, utilized by databases. Section 4.2 describes both relational data model and relational databases in practical usage. Then two database systems convenient for usage as rpmdb engine are picked, described and compared in section 4.2.3.

Chapter 5 deals with the database based on SQL solution. As it was already outlined, is uses information from previous chapters. The design is built methodically based on standard design approach presented in section 4.2.2. Before designing the database itself a conceptual model is introduced and described. Because the database is not entirely based on SQL backend, the other parts are described as well as all connections between them. The beginning of chapter 5 and a part of section 5.2 focus on this.

Finally, chapter 6 is based on two things. First of all deep code analysis of rpmdb was needed. Conclusions of this analysis are discussed in section 6.1. Original idea of the implementation was to make a module which could replace existing database module. This has been proven as ineffective, thus sections 6.3 and 6.4 describe another solutions. The first of them focuses on the proof of concept solution, which has been designed specifically to meet requested assignment and to provide a glimpse how the code using SQL as a backend would look like. Comparison of both solutions is provided in chapter 8. This chapter also evaluates possible contributions of this thesis to future development of RPM.

Chapter 2

RPM management system

General goal of package management system in different Linux distributions was described in chapter 1. It was also stated, that RPM usually refers to a whole architecture working with RPM packages. Figure 2.1 shows diagram of the whole RPM-related architecture.

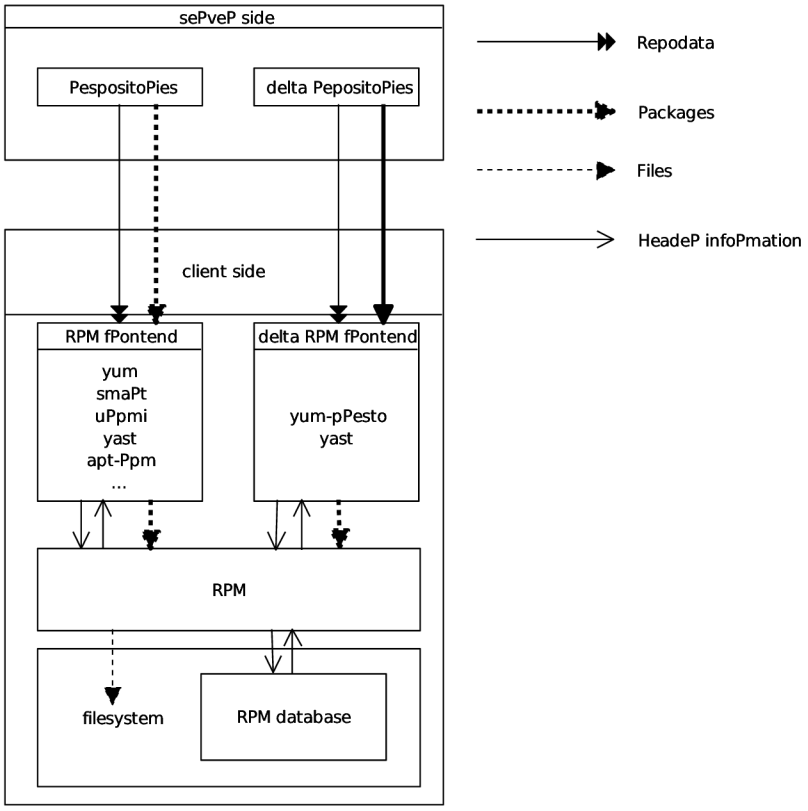


Figure 2.1: Package management system based on RPM

As it is denoted in diagram 2.1, the actual RPM is only one of pieces putting together the whole PM system, even though "RPM" is often used as a name for the entire architecture. To understand the diagram, repositories should be explained first. Repository is a data store containing a number of RPM packages. Packages provided by the repository are compiled (along with some information) into a list represented by compressed SQLite file.

Each client program can have unspecified amount of repositories registered and download programs from there. Repositories can be accessible via different Internet protocols, like *HTTP*, *ftp* but they can be also accessible as common directory in filesystem.

Client programs designed to access repositories will be referred to as RPM frontends. Nowadays, these could be considered the most important part from user perspective. Typical representative of RPM frontends are programs like *yum* (Fedora and Red Hat Enterprise Linux) and *yast* (OpenSUSE, SUSE Linux Enterprise Server/Desktop). RPM frontends have several important parts which basic RPM utilities aren't focused on. First of them is *dependency resolving*. Frontend program keeps the list of packages which it can access in one of known repositories and it keeps the list of installed packages. If user wants to do any operation with packages (either install a new one, upgrade or uninstall one from his system), the dependency solving algorithm is triggered. In case of installation (or upgrade), it decides, which packages to install/upgrade/uninstall along with the program and in case of erasure which packages depend on erased packages and should be removed as well. This is a difference from RPM itself, because RPM only checks the dependencies but it doesn't do anything else, because it has no knowledge about the repositories and their content. To be precise, RPM has some dependency-solving mechanisms implemented, but the code is very old and rather unpleasant from the user point of view – the user would have to download all packages from repository first.

That leads us to the second part which distinguish frontends from RPM itself and that is repository support. Repositories and frontends were designed with cooperation in mind. That is why frontends have advantage here – it is not necessary to download the whole repository, just the package list suffice. Frontend then builds it's own database, when it merges packages from all registered repositories and download only those, which user requests.

Frontends usually have their own databases because RPM database can't contain all the information frontends need. Yum, the most extensively used frontend in Fedora has its own database stored in SQLite format. Frontends using its own databases make significant part of the data redundant because it is already stored in RPM database. But on the other hand this solution has better performance because it isn't necessary to call RPM routines to extract some data,

However calling RPM is necessary during writing operations, because frontends usually don't implement RPM payload handling (installing files, setting their ownership, ...), running scriptlets, RPM database update, etc. These "RPM calls" can be divided in two categories, depending on what is the call target:

- **rpm utilities** – some programs like *beer* directly invoke rpm command and handle only its return code or text output
- **rpmlib** – provides C API and Python bindings. Through these, it is possible to perform operations with RPM database and packages directly

2.1 Content of repository

When optimizing the database, we should know what to optimize for. The basic source of packages which are installed on every Fedora system, are Fedora repository trees "Basic" and "Everything". This section analyzes content of repository for Fedora 12-beta Everything tree. Analysis provides some data useful for better understanding of statistically

"typical" package's composition. Then the same tree has been analyzed for Fedora 9, 10 and 11. This information can provide us prognosis about the future development. The database has lasted about 10 years in its format practically unchanged. It is important that similar situation is expected after 10 years – the database should withstand performance and other requirements with its content for the next 10 years after it is designed.

Table 2.1 shows some statistical properties of packages in different Fedora versions. For package count in repository, the first number denotes total amount of packages, contrary to number in parentheses, which denotes number of packages only for *x86_64* and *noarch*. For various reasons it is possible to have *i686* packages in *x86_64* repository. For example some packages are not available for given architecture, but the original ones can work as well. Another example might be given by multilib parts of the system – it is possible to have the same library installed twice – once for *i686* and once for *x86_64* architecture simultaneously.

		Total	Mean	Median	Mode	Q1	Q3
Fedora 9	Packages	12444 (9881)	-	-	-	-	-
	Header size	284.4 MiB	23.4 kiB	7.6 kiB	-	-	-
	Files	1 806 161	167	16	6	6	67
	Requires	200 583	15	11	4	6	21
	Provides	50 795	4	2	1	1	2
Fedora 10	Packages	11348 (11345)	-	-	-	-	-
	Header size	271.6 MiB	24.5 kiB	7.5 kiB	-	-	-
	Files	1 755 879	154	15	6	6	67
	Requires	178 505	15	11	3	5	20
	Provides	51 944	4	2	2	1	3
Fedora 11	Packages	13195 (13191)	-	-	-	-	-
	Header size	368.4 MiB	28.6 kiB	8 kiB	-	-	-
	Files	1 978 808	149	14	6	6	64
	Requires	218 272	16	12	4	7	21
	Provides	78 117	5	2	2	2	3
Fedora 12	Packages	19120 (15343)	-	-	-	-	-
	Header size	492.1 MiB	26.4 kiB	8 kiB	-	-	-
	Files	2 489 520	130	13	6	6	57
	Requires	339 129	17	13	5	8	23
	Provides	113 017	5	3	2	2	4

Table 2.1: Statistical properties of Fedora packages

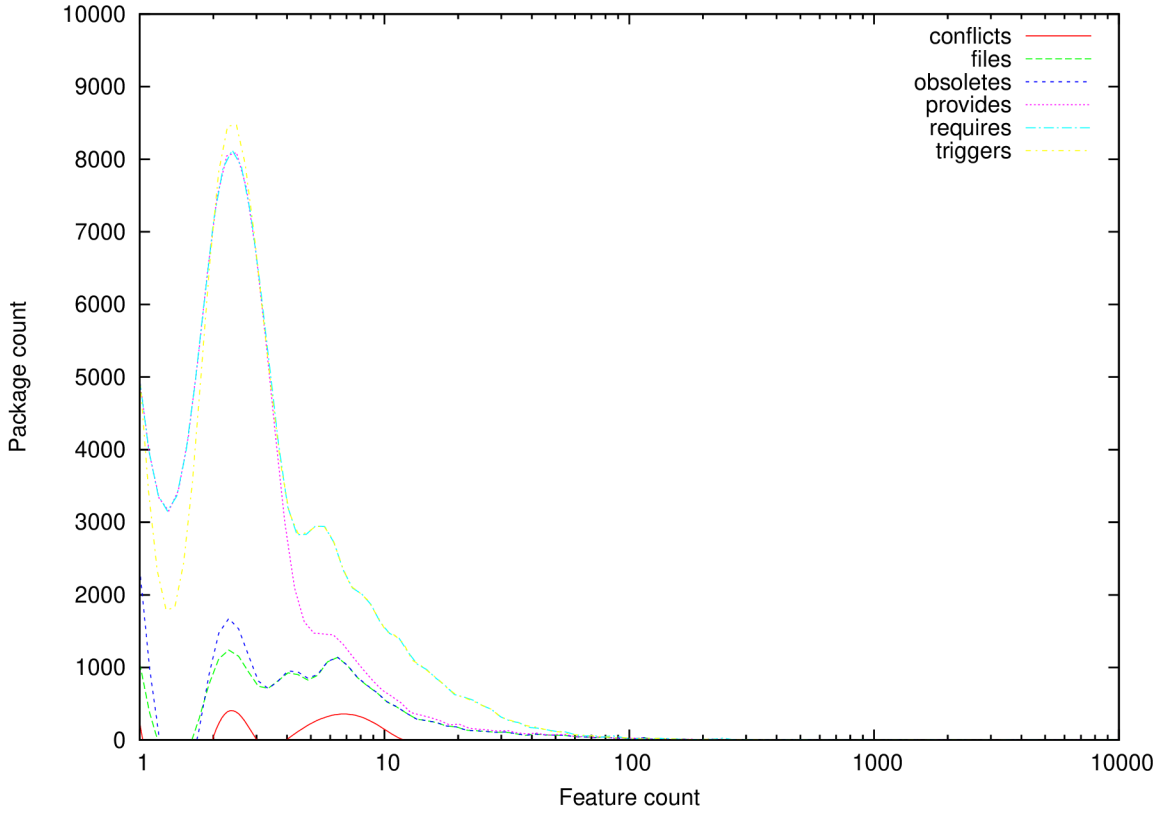


Figure 2.2: Dependency of package count on given property count

Table 2.1 provides some statistical information about recent Fedora versions and packages included in their basic repository trees. Mean values in the table can contain error caused by integers used in scripts generating statistics. Maximal margin of this error is ± 1.

Graph 2.2 shows approximated property graph – it shows how many packages have given amount of records in one of `Provides`, `Requires`, `Files`, `Obsoletes` or `Conflicts` RPM tags. It is important to realize that graph is only approximation to give an idea about how package features are distributed. For the sake of lucidity some peak values were ignored on both X and Y axes. Again this step was taken because there was no need for the graph to be precise.

Various versions of Fedora were included in the statistics to estimate future development. The statistics were made from the whole repository (not just from packages for given architecture), because it is usually possible to install multiple package versions (to be more precise one package version, but for multiple architectures) on one machine (i.e. it is possible to install both glibc.x86_64, glibc.i686 and glibc.i586 on one system). As it was stated before, the whole repository was included to estimate probable top margin of what the database will have to withstand. It is possible to object that user most likely won't have all packages from repository installed. That is indeed true, but on the other side, user can have multiple repositories enabled and have large portion of packages provided by them installed.

Now to the results and conclusions of package statistics. What we can see in both graph 2.2 and table 2.1 is that most of the packages are composed of very small amount of files (about 52% of them has 13 or less files). Situation for provided and required features is

very similar. What can seem strange is mean value of files contained in package, which is ten time greater than mean value. Here it is important to realize that mean value is caused by some packages which have up to thirty thousand files included. It is available to see in graph 2.2 that amount of packages containing more than 300 hundred files is very small (in fact, not even 5% of packages have more than 500 files). Despite the small amount of such packages, they influence the mean value significantly, that's why the focus is laid on median value here. As for Requires and Provides, there is nothing highly unusual there, mean and median value are similar. From these statistics we can estimate a **typical package representation** with these statistics:

- **Files:** 6-13
- **Provides:** 3-5
- **Requires:** 13-17
- **Header size:** 8-26 kiB

And now to the possible future development. The first obvious and important things are package count in the repository and their header size. These two values indicate how the database shall grow in next few years. Figure 2.3 shows their development.

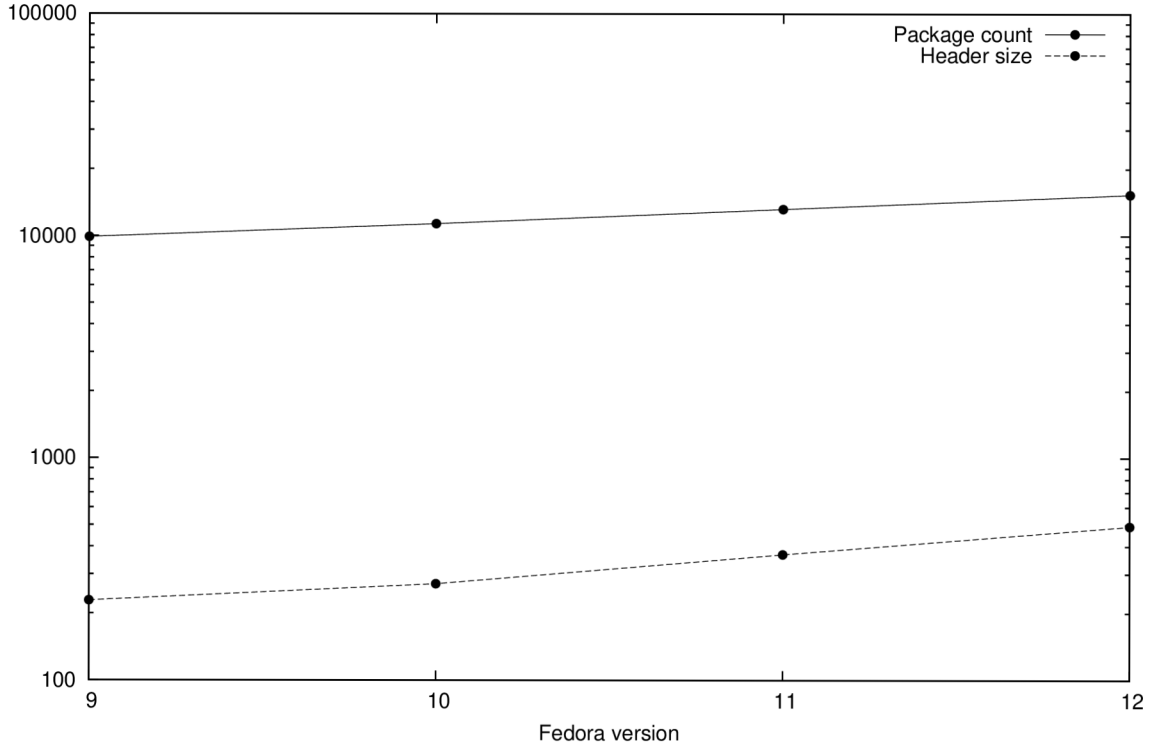


Figure 2.3: Package count and metadata size development in recent Fedora versions

It shows that number of distinct packages (only those for x86_64 and not their counterparts for different architectures) increases in each version by a little over 2000 packages. It is possible to expect growth similar to graph 2.3 in the future, although there might be a top border, because of different reasons – for example smaller number of programs with

new functionality. Another thing shown in graph 2.3 is total metadata size in all packages. Besides package count, metadata in every package grow as well, because there are things like changelog contained in every package header. Since both package count and header size grow linearly, the resulting metadata growth should be polynomial (total metadata size is counted as $\sum_{i=0}^{Pkg\_cnt} Header\_size$). In the graph it is possible to see a comparison between metadata size and package count growth. It is say about the same despite previously stated fact about metadata growth. That can be explained by division on packages. Some packages are split into two or more, which leads to package count increase, but metadata size stays more or less the same.

Based on graph 2.3, it is possible to estimate approximated metadata size after ten years. By that time, it is possible that common database will have around 500 MiB and in some extreme cases (such as all packages from repository installed) it could easily reach 1 GiB. All these numbers consider only the primary part of the database without secondary indices (in current solution that means only a size of `Packages` database).

2.2 RPM architecture

RPM itself is an extensive system, which contains several parts. Diagram 2.4 shows the main grouping.

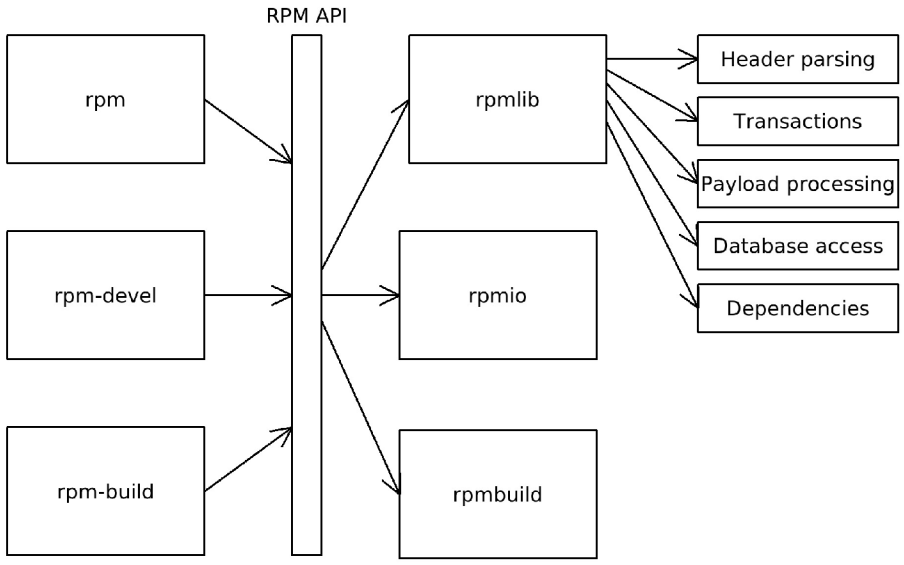


Figure 2.4: RPM system composition

It is possible to divide RPM into two main groups - libraries and programs built on those libraries. We will start with programs. There are three RPM-related packages. More exactly they are directly related to RPM. Some other packages exist (e.g. for inspecting RPM packages and for supporting the development of new packages). Following list shows these three directly RPM-related packages and programs they provide.

- **rpm**
  - *rpm* is a utility for general manipulation with packages and for querying the RPM database
  - *rpm2cpio* can extract payload from RPM archive
  - *rpmdb* is a tool for database operations
  - *rpmverify* provides a subset of rpm-provided operations, particularly it supports verifying. In general, that is operation, when files installed/owned by the package are compared with information in the database (or RPM header) and eventual inconsistencies are reported.
- **rpm-build**
  - *rpmbuild* is a tool for building rpm packages from sources
- **rpm-devel**
  - *rpmgraph* displays package dependency graph using graphviz

The programs listed above utilize different parts of RPM API, which can be divided into three libraries. These libraries along with their description are in the following list:

- **rpmlib** is the main RPM library. It includes core operation set for every part of RPM. It cooperates closely with rpmio. Here follows list of some important parts:
  - *Header parsing* – reading, loading and processing package headers. Conversion from and to format used in rpmdb and RPM packages is also part of this module.
  - *Transactions* – Every operation (or set of operations) is represented as a transaction. This module handles the transactions, their iterators and elements.
  - *Package payload processing* – supporting functions and file state machine for handling cpio format
  - *Database access* – this is the subsystem this thesis is focused on. It contains all the functions used to communicate with rpmdb, also some data conversions are here.
  - *Dependency related* – dependencies, problems in them and added/available package list
- **rpmio** is supporting layer for accessing files by rpmlib (local or those available through network)
- **rpmbuild** – library providing general functions for building and composing rpm packages

2.3 Package structure

Content of RPM packages can be divided into two separate groups. Because packages carry software, their main content is composed of files, which are copied to filesystem during the installation. Although the second part is not important from the user's perspective it's the most important for RPM database. The second part contains supporting information. One

chunk of this information is composed of scriptlets – small pieces of code (usually written as shell scripts) which are executed at given point of package (un)installation. Package can contain these scriptlets: `%pre`, `%post`, `%preun`, `%postun`, `%triggerin`, `%triggerun` and `%triggerpostun`.

Following text shows the order of scriptlets in the most complex operation with RPM package, which is upgrade (or downgrade – operation is practically the same). Upgrade consists of installation of new version followed by uninstallation of old version(s):

1. `%pre` of the new package
2. *installation of new package's files*
3. `%post` of the new package
4. `%triggerin` of other packages
5. `%triggerin` of the installed package (package version respectively)
6. `%triggerun` of the uninstalled package (package version respectively)
7. `%triggerun` of other package
8. `%preun` of uninstalled package
9. *uninstallation of old package's files which aren't part of the new package*
10. `%postun` of the uninstalled package
11. `%triggerpostun` of uninstalled package (package version respectively)
12. `%triggerpostun` of other packages

When simpler operations take place, certain scriptlets are left out. As it is obvious, during installation the chain ends with install triggers and uninstallation begins with uninstall triggers.

Scriptlets obviously need to be saved in the database, especially those, which are related with uninstallation, because they can be executed after the original package file was removed. The same case applies for triggers.

Triggers are special scriptlets, because they can be executed even when package they are part of isn't subject of any operation. That means they can be triggered by any installed, uninstalled or upgraded package handled by RPM library. This implies that triggers must be easily accessible using name of triggering package (and possibly version) as the lookup key.

The remaining part of RPM file consists of so called tags. These tags create information about package and it's properties. From strict point of view, scriptlets are tags as well [20], even though their purpose is little bit different from the other tags. More details about how RPM tags are stored are disclosed in section 2.4. Complete list of tags can be found in source code of RPM, specifically in file `lib/rpmtag.h`. Although it is not completely up-to-date, RPM guide provides the list of most important ones for package maintainers, describing both their value and purpose (alias usage). Some of them are significant for this thesis. Most of these important tags are important for current rpmdb as well – because they are commonly used, there are dedicated databases for them in order to boost performance. Detailed description is in section 2.5. Important tags are concretely:

- *provides requires conflicts obsoletes* – these tags are used for solving dependencies, which is frequent and important routine in every packaging system including RPM. Each of these tags has three parts – name (mandatory), version (optional) and flags (>= = <= < >, optional)
- *NVR*, which stands for Name-Version-Release is internally needed for large subset of operations. It is used as unique identification for every package. Its extended form is sometimes used and it's called NEVRA (Name-Version-Epoch-Release-Architecture).

2.4 Package file format

This section describes how the data are stored in RPM file. An overview of RPM package is offered by figure 2.5.

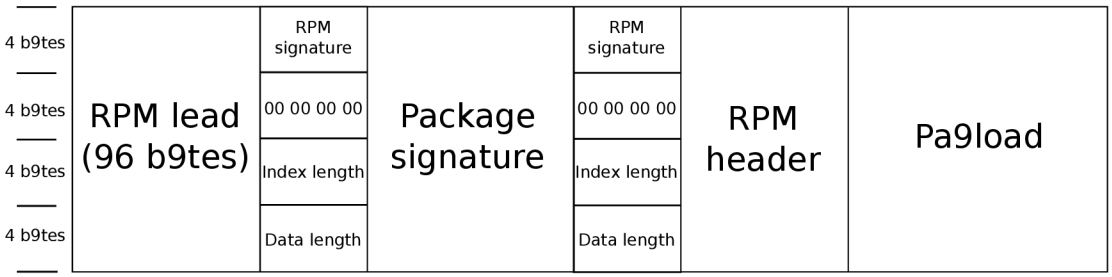


Figure 2.5: RPM package structure (top to bottom, left to right).

As the figure shows, the first part of RPM package is *rpmlead* structure followed by *signature*. The lead is deprecated and it remains for backward compatibility[20], it isn't used otherwise. Therefore when reading RPM file, it is possible to skip first 96 bytes `(88+4*sizeof(short)+padding)`, which the RPM lead is created of (plus padding). The following two parts are signature and header. They have the same format:

- 4 bytes of rpm magic number signature
- 4 zero bytes
- 4 bytes network-byte-ordered index size (further referred to as l_i)
- 4 bytes network-byte-ordered data length (further referred to as l_d)
- index structure array (1 structure containing some general package metadata and 1 index structure for each tag present in the package)
- byte array with the data index structures point to

RPM magic number indicates that file is valid RPM package. Following 2 numbers are used to determine length of subsequent data as $l_i * sizeof(struct\ entryInfo\_s) + l_d$. Header itself is created by the data. It is byte string of values separated by zero bytes. To know which value is which, index is created on the top of these data. Index is an array of below described structures[20]. Each of these structures identifies a chunk of data as one of RPM

tags. It contains information about tag's data type and which tag it is. Then it contains an offset of the data – that is position of the first byte of data from the beginning of data blob. As it was already written, the end of data is designated by zero byte.

```
struct entryInfo_s {
    rpmTag tag;          /*!< Tag identifier. */
    rpmTagType type;     /*!< Tag data type. */
    int32_t offset;      /*!< Offset into data segment. */
    rpm_count_t count;   /*!< Number of tag elements. */
};
```

The data in RPM signature contain maximally 32 index records (max. 8129 bytes of data payload respectively). Signature is used to describe the RPM file (as in the package container), not the package itself, that is why the limitation is present – there is not much information needed to describe the file. For this thesis RPM signature isn't important. Contrary to signature, header is the most important part, because it contains all package's metadata. Also the format which was described above is important, because the same format is used in RPM database. The data in header are more important, since they are stored in rpmdb.

The rest of package creates a file payload. Because it is not important for further chapters, it won't be analyzed closer.

2.5 Database structure

Data files are by default stored in `/var/lib/rpm` directory. The basic part of the database is file `Packages`, which contains all the information about installed packages, and thus it is the primary RPM database. As it was written in section 2.4, it uses the same format as is used in RPM packages (obviously without the first 8 bytes – the magic number and zeros). Position of the package metadata in primary database (and also its lookup key value) will be in the rest of this section referred to as *installid*.

The `Installtid` file contains install transaction times (i.e. when individual packages were installed), as well as IDs of those transactions. The other important files in the directory are indices of the primary database. Based on this relation and the fact they are also Berkeley databases, they can be referred to as *secondary databases*. They are designed as reverse databases to `Packages`. It has *installid* as the key and the rest of the data as value. On the other hand every secondary database has one of RPM tags as a key and *installid* as a value. More specific information is listed below. [15] Each secondary database file uses its own BDB index (Hash or B-Tree). BDB index type for every secondary database is listed below as well.

Basenames (hash) File name (not a path, only a name) is a key and values contain pairs of `(installid,basenameindex)`. This index is very closely related to *Dirnames*

Conflictname (hash) Name of conflicting package is a key here(RPM tag *CONFLICTNAME*), pairs of `(installid,conflictindex)` then create values

Dirnames (B-Tree) Index containing paths to files listed in *Basenames*. Path is a key and values are `(installid,dirindex)`

Group Valid name of a group is a key (according to file `GROUPS` in a directory with RPM documentation, e.g. `/usr/share/doc/rpm-4.6.1/`) values are created by couples: `(installid,0)`

Name (Hash) Package name is here as a key and values are: `(installid,0)`

Providename (Hash) As same as *Conflictname*, only uses RPM tag *PROVIDES*

Provideversion (BTree) Contains versions of packages listed in corresponding records of *Providename* as a value

Requirename (Hash) Index is corresponding with *Conflictname* and *Providename*

Requireversion (BTree) Index corresponds with *Provideversion*

Triggername (Hash) Key is unique name of a package trigger. Values contain couples of `(installid,triggerindex)`

2.6 Database backend

RPM is based on *Oracle Berkeley DB* engine. It is open source database library, which is linked with application and thus mapped to memory space of a process which uses it. That can be a huge advantage in terms of memory consumption. It also means that DB4 doesn't provide standalone server, but application providing it can be implemented using this library. For example some older version of MySQL were offering BDB as storage engine.[10]

Main features of this backend are:

- ACID semantics involving writing locks, logging, data restoring after corruption and multiple operations in one transaction
- page cache management, including I/O handling
- several index types (B-trees, hash tables, queues, numbered lists)
- no restriction for stored data (they are represented only as a byte string of certain length)

BDB uses simple data format. Every database is contained of key and data values. Both of these are represented as a structure containing byte string and the length of this string. No other information is contained, therefore understanding the data requires explicit knowledge how the data are organized. That was described in previous sections. Every BDB database has its own index. That is used for example to look up value for given key. We need to know the index type, otherwise it won't be possible to open database file. [17]

Utilizing Berkeley DB brings both advantages and disadvantages to RPM. The main advantage is extensibility of stored data. This feature comes from used data format and how the data are stored. In comparison with relational database, which in its normal form requires data to be atomic, BDB allows the data to have arbitrary form. This also applies for key values, which can be thought of as another advantage.

As for disadvantages. The largest one is given by the data format. Because the database has no idea neither what the data represent nor what is their type, we have to handle all the data ourselves. Another disadvantage resulting from the data format is difficult implementation of complex queries. Because key is basically a binary blob and it is permitted to have only one key per database, it is impossible to execute complex queries like searching a package by multiple parameters (e.g. name and architecture) directly. This can be worked around for example by creating secondary databases as indices – this basically emulates possibility of having multiple keys for one database for a certain amount of performance.

DB4 features are important for RPM database. One of them can be determined using data in section 2.1 and measured database size. In Fedora 9, it's 318 MiB, in F10 it's 302 MiB. In F11 and F12 it rises significantly to 417 MiB (562 MiB respectively). For better picture, these values are displayed on figure 2.6. Using these resources, it is possible to count database overhead. Because it certainly depends on the data that are inserted, it was measured on rpmdb. Because all packages were installed at once, the overhead was at its minimal possible level. To be certain, it is possible to run `rpm --rebuilddb` command to optimize RPM database. On testing machine after rebuild of the database, there is no significant change in its size (reduction only in a magnitude of kilobytes). However when using RPM for some time in common environment, database overhead grows – on testing machine which has been used for several months without database cleanup, rpmdb was reduced from 85 MiB to 53 MiB only by performing rebuild.

Graph 2.6 shows both the database size and overhead. The database size is made by total metadata size and overhead size. Both parts are displayed in figure 2.6. From this graph we can read, that database overhead is between ten and twenty percent. Particular overhead is 33.6 MiB for F9, 30.4 for F10, 48.6 MiB for F11 and 69.9 MiB for F12.

It is possible to calculate the overhead more precisely. From previously collected data, it is possible to construct a graph displaying the overhead dependency on database size. We can calculate a slope of line between the first and the last points of this graph. Example of such graph is displayed by figure 2.7 (the graph is only illustrative, although it has been constructed from real data). The slope is calculated on the line between the first and the last point on the graph and its value is 0.18, which means the overhead is 18%.

This approximation isn't good enough, because it would mean that for small databases the overhead is negative. To make it more accurate, a very small database is needed. We create one and fill it with some packages. An example is given below. This sample database is 2173 kiB large. Actual size of package metadata is 1820.25 kiB, which makes overhead 355.75 kiB. Taking this into consideration, the overhead is 16%. If we wanted to be precise, the growth would have slightly polynomial character. To display that, much more data would be required. Considering the exact percentage isn't goal of this thesis and we need it only to have a rough number for comparison with potential new solution, out current approximation is sufficient for us.

```
# An example of creating and filling test database
rpm --initdb --dbpath /path/to/tmp/db
rpm -i --nodeps --justdb --dbpath /path/to/tmp/db /path/to/local/repo/xorg-*
```

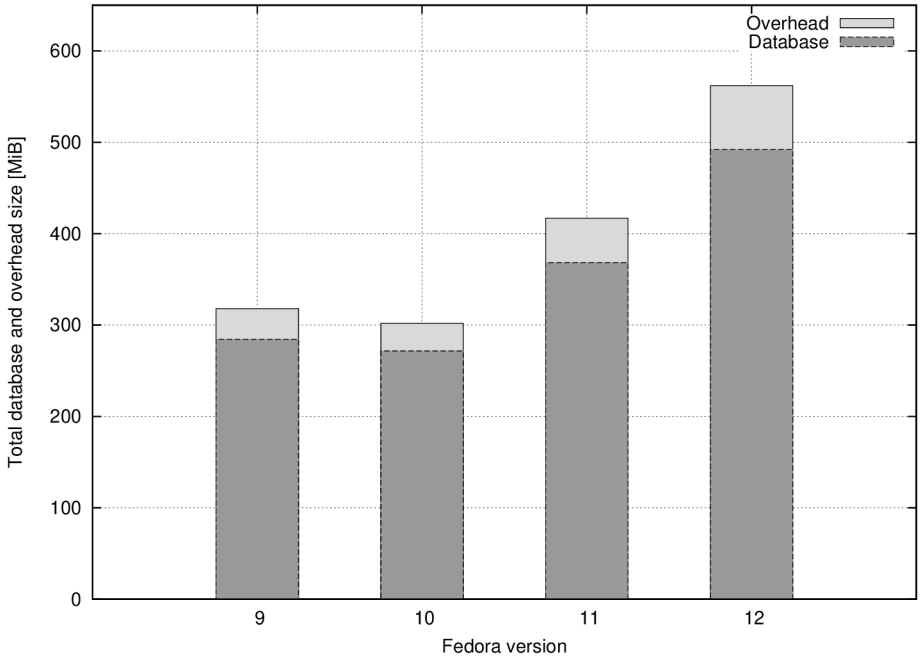


Figure 2.6: Database size in different Fedora version

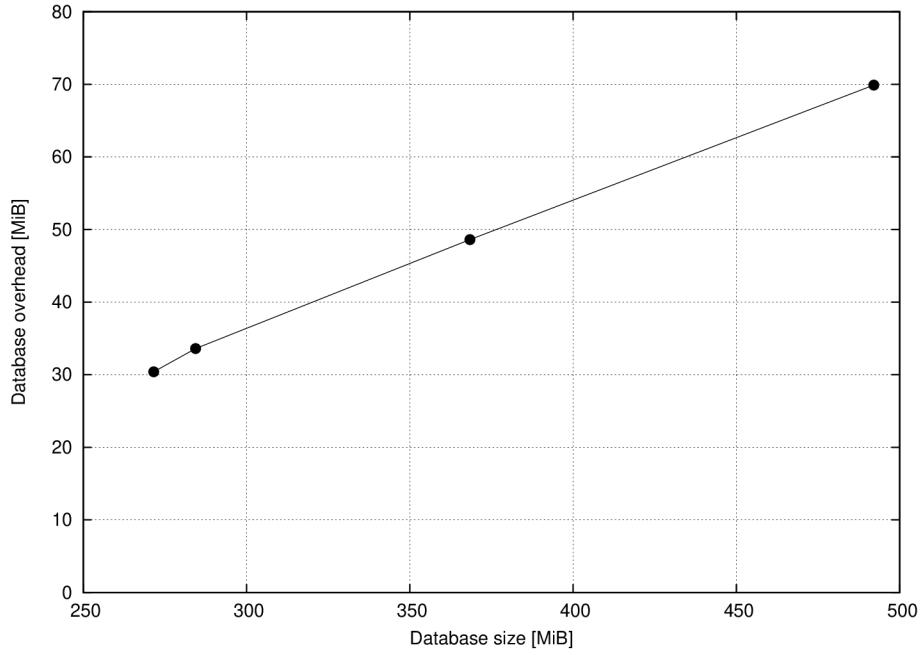


Figure 2.7: Dependency of database overhead on database size

Chapter 3

RPM database analysis

3.1 Performance profiling

Several profiling test were run on order to determine where the weakest spot of RPM is. Profiling graphs are all on attached DVD, because of their large proportions. Tests have been run on Fedora 11. There were 2 types of tests. First of them was on normal workstation – that means only commonly used packages were installed, database size was approximately 80 MiB (database further designated as *normal-db*). The second type was on full database (over 400 MiB, further designated as *full-db*). Description of tests follows:

At first a blank database was created, then all packages from Everything repository of F11 were installed and finally some queries were ran. Query formatting was used instead of parameters like `--list` because of this parameter's behavior. In it's implementation it calls some unnecessary functions which leads to major distortion of results. In the end, the database was deleted:

```
rpm --initdb --dbpath /path/to/db
rpm -i --justdb --nofiles --nodeps --dbpath /path/to/db /path/to/repo/*
rpm -q --info --dbpath /path/to/db rpm
rpm -qa --dbpath /path/to/db
rpm -qa --qf "[%{FILENAMES}\n]" --dbpath /path/to/db
rpm -qa --qf "[%{REQUIRENAME}\n]" --dbpath /path/to/db
rpm -qa --qf "[%{PROVIDENAME}\n]" --dbpath /path/to/db
rpm -e --nodeps --justdb --dbpath /path/to/db <list-of-all-packages-in-db>
```

Then the whole process was repeated with selected packages. List of those packages was derived from list of packages installed on author's working computer. That list is considered to be a representative pattern for normal installation. A set of scripts used to generate the list and subsequently an install script is present on attached DVD.

Before executing any of listed operations a following test took place to figure out if disk cache has any affect on profiling results. The script (at least the synchronizing and cache-flushing part) should be run by root. Graphs for cache available and flushed are present on attached DVD. Both graphs are the same when displaying the the percentage of consumed time. When using absolute numbers (ticks of CPU), they are slightly different, but the difference isn't big. From these findings it's safe to assume flushing the cache doesn't affect profiling.

```
rpm -qa --list --dbpath /path/to/db # create cache
valgrind --tool=callgrind rpm -qa --list --dbpath /path/to/db
sync
for i in 1 2 3; do
  echo $i > /proc/sys/vm/drop_caches
done
valgrind --tool=callgrind rpm -qa --list --dbpath /path/to/db
```

In test result, some of the routines are marked only with their address in memory. Those routines are outside of rpm, rpmlib and db4 library – that means they are not much important, hence we won't occupy ourselves with them any more.

What we can see from graphs is how much of execution time is spent reading data from the database and parsing them (or in case of writing operations how much time does it take to write the data to database).

Using common sense it is possible to divide RPM operations into 3 basic operation modes. We start with read and write division. Write operations can be further divided to *install* and *remove*. Within each of these categories we can even further distinguish simple and iterative operations (repeated over large amount of packages).

The first operation is installation. Data were gathered by installing all packages in Fedora repository and by installing only subset of these. Bottleneck during iterative installation is indexing of files and directories. They are added to a *list of available features* and this list is sorted for each installed package. Comparing full-db and normal-db installation it is possible to see that relative portion of time spent in sorting grows with number of installed packages. As for data format and storage related operations: DB4 related operations take about 10% of CPU time. This can't be reduced much, since all package information has to be stored in the database. The only possible reduction here can be done by deciding which information from package header is not relevant and thus isn't neccessary in rpmdb. Example of such information may be complete changelog. What is more interesting is conversion of package ID sets from stored to working format, which takes about 17.5% of CPU time. This amount of time is significantly influenced by data format. Fundamental issue here is byte order detection and correction and cardinality of tuples stored in set (1 or 2 numbers in tuplet). Because the database will most likely not available for transfer to another machine, not yet to another architecture, this can be easily avoided. According to one of RPM developers, the byte-order detection and conversion might be present because BDB stores the data in network byte order. In that case switching backend database would help. Of course, the new backend would have to do the conversion itself – or even better no conversion should be performed at all.

Erasing operations have one great bottleneck and that is cleaning of package-ID-sets. These sets are contained in every secondary index, as was described in section 2.5. That means erasing all package records (there can be large amount of them in some cases) has to be done for each secondary index. Erasing contains quicksorting and subsequent binary searching in set. These two functions take most of the CPU time during erasing operations. Rpmdb can help reducing this time by providing *tertiary internal index* or similar mechanism implementing more data caching. Other significant profiling properties (and subsequently possibilities of their affection) are basically the same as they were for installation.

When querying the database the largest time consumer is hash checking. That however can be disabled by `--nodigest --nosignature` options given to rpm. When we consider

data provided by default run as primary (i.e. hash checking is turned on), we can state that reading from the database takes about 1.5% and parsing the data takes about 10%. We will return to these values, now to the results when hashing is turned off. In that scenario, we have several major time consumers identified. First of them is previously mentioned header parsing in form of `headerLoad` function. The second largest is header formatting for output, the third is loading data from rpmdb and the fourth is loading RPM macros. Because they are not rpmdb related, we can omit loading macros and formatting. The result confirms that there are only those two above mentioned operations which could improve query performance and are related to rpmdb. Now back to the first results. As in previous cases loading the whole header is the most significant performance retarder related to rpmdb. It is responsible for both above mentioned time consumers. Again this could be solved if we didn't load all data from rpmdb at once. At the other side, it is important to mention that loading the whole header has one advantage and that is performance stability when querying for various data. Since we have all the information about header available, the only difference is parsing them to output.

3.2 Other performance tests

Major slowdown of RPM operations is caused by calling fsync on databases, especially the secondary ones. It has been proven that removing fsync calls raises the performance significantly[14]. This could be upgraded either by switching backend or by change in writing style to current rpmdb. Here is an example of such change (when writing more items to database):

1. Write data to Packages file for every package
2. Sync after each of the packages
3. Set the flag to *"out of sync"* state
4. Perform all changes in secondary indices for all packages
5. Call sync on all files
6. Set the flag to *"synced"* state

Another performance test shows dependency on disk cache and overall querying speed of rpmdb. Two tests were run, both were lookup-type tests. One of them used secondary database as index, another one didn't, because the secondary database it would need isn't available. This is how the test was executed:

1. A list of packages in the database is retrieved
2. Several packages are randomly picked from the list.
3. For each of these packages a test is run for a given number of times
   (a) Drop memory caches and call sync to write all unwritten data do disk
   (b) Run a lookup of the package for a given number of times, use the `query` program which prints the time which lookup took

4. Calculate mean value for each one of runs from all test that were held (for every package several tests and there were multiple packages)

5. The result is a list of mean values for every run from cache-dropping point

As an input to this test, we need a number of packages that are chosen from the list, a number of times every package is probed and how many iterations from dropping the cache should be run. This test is more closely described by python script which is on attached DVD (in directory `3.2`).

Because absolute numbers are results of this test, it is important to have a reference machine described. The test was performed on a machine with Fedora 11 installed. The configuration of important parts is described below. All test were performed while nothing else was running on the machine except X server and basic system daemons.

```
Hewlett Packard xw4600 Workstation
Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz, 6MB L2 cache
4GB DDR2 RAM @ 800MHz, SAMSUNG SpinPoint F1 DT series 500GB
```

Program `query` was specially written to perform queries on rpmdb. Queries are as simple as they could be to eliminate influence of non-measured elements as much as possible. The program is present on attached DVD (in directory `3.2/query`). It supports two basic modes:

In normal mode the whole operation is measured, including memory allocation, etc. On the other hand in optimized mode only lookup itself is measured. That means for example opening/closing the database and memory management is omitted.

Work in each mode is taken into consideration and it is presented in graphs 3.1 and 3.2.

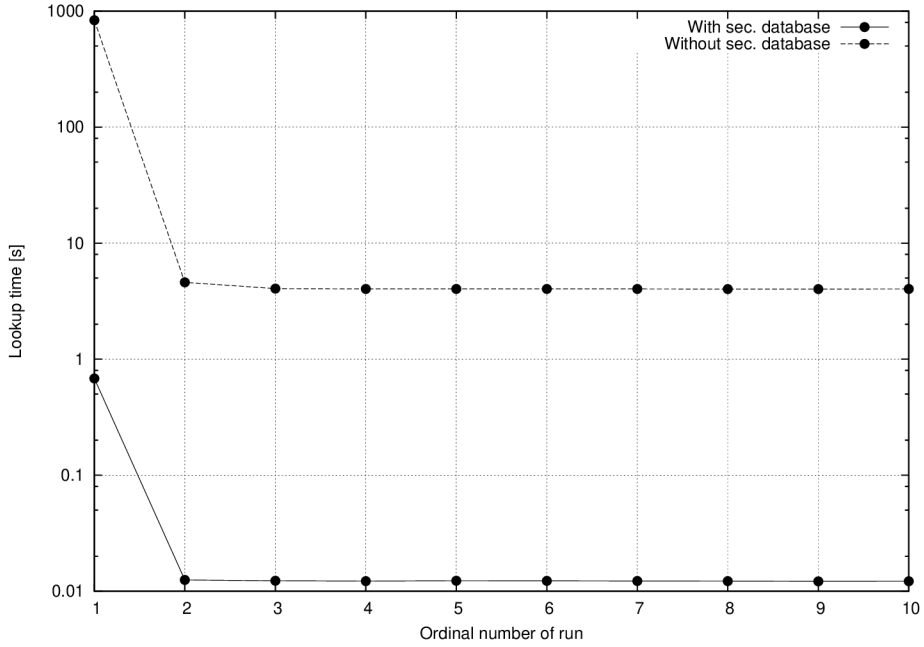


Figure 3.1: Dependency of rpmdb file cache

Figure 3.1 displays two graphs. In both cases their first run is significantly influenced by disk cache and the others are relatively stable. The first run for indexed lookup took 0.68 s.

For non-indexed run it was 833.598 s. Times of the following runs are specified below.

Besides cache influence on query, figure 3.1 shows the difference between using and not using secondary databases as index. The result is not surprising: when using secondary database, lookup time floats between 12 ms and 13 ms. Lookup time of query not using the secondary database is between 4.01 s and 4.03 s. Simple calculation shows that secondary database can speed up the lookup operation more than 300 times.

Another performance comparison is described by figure 3.2. It is based on the same program, but it measures only search results, not opening and closing the databases. Some optimizations like exclusion of memory allocation for results have been done as well. The first result from this run shows that opening the database when cache isn't loaded is significant (more than 90% of indexed lookup and about 10 s when using non-indexed lookup). And for the rest of runs result show that opening and closing the database is important when we used secondary indices (in the case of not using them, the difference is the same as it is in the first case, but considering the total lookup time they are not as significant). To be more precise, these optimized runs took approximately 3 ms, which means 75% boost. This can be used when implementing database related algorithms – we should focus on opening and closing the database only once per transaction set and not for example once per transaction.

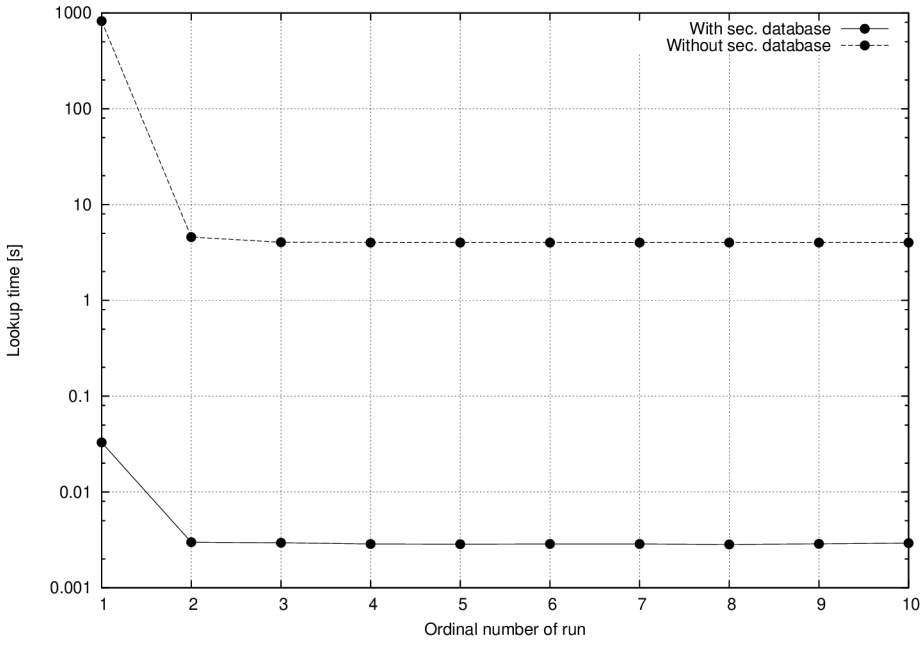


Figure 3.2: Dependency of rpmdb file cache (lookup time only)

3.3 Shortcomings summary

From previous text the obvious and largest shortcoming is a large data blob loaded and then parsed to gather information. The first problem with this is that it generates significant I/O load when inspecting a large number of packages, even if only a small chunk of the data is actually used. For example when listing NVR of all packages, it is necessary to load the whole header for each package and parse it. That subsequently unnecessarily creates also

CPU load necessary for parsing the whole header. Loading the whole header also invokes its checksum counting and verifying and as section 3.1 describes, that leads to major CPU time consumption. The second problem with loading the whole header is that at that point the database is barely utilized and it would be possible to reduce `Packages` database to a simple blob, which would be mapped to memory using `mmap()`.

There are some other performance shortcomings, but there are not as significant and were already described, so let's skip them and focus on another design aspect and that is transactional behavior. The main issue here which needs to be solved can be simply represented by a situation when package is installed, but in the middle of installation something happens (power failure, system freeze, ...) which causes the operation to be interrupted. There are some aspects which bring more complication. The first one is difference between the filesystem and database. Contrary to database, the filesystem state is not fully revertible in case of failure. Then of course there is a complication in a form of scriptlets. For the operation to be fully revertible, it is absolutely necessary to keep track about which part of a scriptlet is just executed.

Chapter 4

Possible design approaches

4.1 Other packaging managers

There is a number of package managing systems, which have at least some degree of similarity with RPM. These package managers are not only domain of GNU/Linux systems, other UNIX-based systems use them as well. On the other hand not all Linux distributions use package managers, some of them have only very basic packaging systems and some of them even don't have anything like this and use only self-compiled software (e.g. Linux From Scratch project).

Particular PMs specified below won't be subjects of copying, but rather subjects of inspiration, because some of their features and approaches are interesting and can be certainly used during design phase. These features will be described in following text.

One group of PM systems use SQL engines and databases. The first packaging system in this group is **Conary**, used in ForesightLinux and developed by rPath Inc. The system is written in python and can utilize several SQL engines as backends. [18] Conary was originally developed by a number of an early Red Hat developers and it focuses on several features:

- uniting local database handling with repository related work
- repository content should be more than just a bunch of files and a bit of metadata
- advanced package version control including branches should be in place
- scriptlets should be eliminated, since they represent a major obstacle for package portability

Fundamental concepts of this packaging system are completely different from RPM: the whole system is not package-oriented, but file-oriented. Packages and subpackages (both called "troves") are basically only containers for one or more files. Also version control is present in every part of the system. What is also interesting is that scriptlets known from RPM were replaced by *tags* - a mechanism, which provides a large degree of portability. Tags are basically references to scriptlets. The main difference is that scriptlets are not a part of package, but rather are a part of the packaging system, hence one scriptlet can have many versions for different distributions. [13] Disadvantage of this approach is that it's not that scalable – the number of officially distributed scriptlets is limited and there are some packages which need special actions to be taken before or after operation with them. For those packages tags coexisting with some additional scriptlets might be an option.

Another member of this group is *yum*, RPM frontend used in Fedora and other distributions. This package manager uses SQLite databases downloaded from repositories extended with some simple text data. There are three databases. The first one is basically copy of part of rpmdb. It has table "packages" and few other tables used for dependency resolving. The second one is a file pool. It contains all the files and their relations to particular packages. And the third one contains all other data. It is even possible that this is the database, where all the data, which aren't important enough to be in previous two databases but are important enough to be in the database, are kept. [19]

RPM isn't the only system using Berkeley DB. Pardus is a Turkish national Linux distribution. Its package management system, *pisi*, uses this database as well. Pisi is one of above mentioned system which combine package database handling with repository related part. Its database consist of several separated parts. Because pisi manages repositories, some data are stored in the database about them. Mentionable concept is that pisi stores information about all the packages in registered repositories. On this set of packages is based another database, which stores information about their state in the system (whether they are installed or not, which version is installed, ...).[3]

Most of packaging systems use plain text to store package information, rather than some kind of database engine. This approach has one major advantage and that is human readability. This advantage however can have its price and that is performance. It is difficult to compare two package managers with goal of comparing their backends, because every PM has its own way of handling a command and most performance differences are caused by exactly this. For example installation will be influenced by hash calculating, data structures used, etc. But in certain cases it is very likely, if not certain, that database engine optimized for speed will have much better performance when used right. This will apply in cases that a single file is used to store information. In a case of entire directory structure, it could even prove to be more efficient than database systems. Both approaches will be described later.

Probably the most known packaging system using plain text files is *dpkg*, used in Debian and its derivatives. It stores its database in `/var/lib/dpkg`. The main part of it is file `available`, which contains list of all installed packages. This list is in extensive format – the record for each package contains a major part of its header (name, description, dependencies, size, ...). Records are delimited with blank line, like it is done in HTTP protocol. Similar to this file is `status` file, which contains practically the same information complemented with its current and future status, used for example when package is installed, but not yet configured. The verbosity of these files makes their size significant, although not as much when compared with RPM, which has even more verbose database `Packages`. To handle file this large with performance in mind, the developers use mmap to load the whole file into memory and then work with it. Actually this approach is used in more programs as it is quite fast. APT, the frontend of dpkg, uses the same data storage format as dpkg does, complemented with some other information like auto-installed flag or list of files and checksums of database downloaded from repositories.

Another PM using text files is *ABS*, used in Arch Linux. Following format is very similar to that used by *portage*, utilized by Gentoo Linux distribution. Both formats originate in that used by BSD packaging manager called *ports*.[7, 2] ABS – the arch build system – uses directory tree to store information about installed packages. In this tree, every package has its own directory, which contains all of packages' stored metadata. [5] There are three basic files: `depends`, `desc` and `files`. The first one contains list of packages which current package depends on and the last one contains list of all files owned by package. The

`desc` file contains the larges amount of information, although the information is not that significant. It contains some records similar to those which are in *spec files* – files describing the package and how to build it, used in RPM. Possible contents of this file include name, version, description, license, etc. . This (or similar) format of data stored on hard drive is widely used on variety of Linux distributions and other operating systems.

When investigating PM systems, it is possible to notice some differences between RPM and a major part of others. One of them are dependencies. Almost all PM system use package-level dependencies. Back in section 3.1, it was outlined, that one of major problems of RPM is a large amount of data it is stored in the database. Here a reduction could start, but it would need a drastic change in RPM packaging policy. By making dependencies package-focused instead of feature and file focused, it is possible to achieve significant reduction of packaged data.

4.2 Relational databases

4.2.1 Relational data model

Relational databases are one of the mostly wide used database engines. One specialty about them is that in history, they were implemented first and the supporting mathematical thesis (or rather model) was developed and published afterwards. This data model is based on mathematical relations. Relational database store the data in tables. It is often thought, that relational database are relational because there are semantic relations between tables. But when knowing the supporting mathematical model, it is quite simple to find out, that this is not true, because in the data model, tables themselves represent mathematical relations. The transition from mathematical model to practical design will be explained below.

Because relational data model is one of database models, it has to contain a few specific parts:

- definition of logical structure
- definition of integrity constraints
- formally defined query language

The first one is related with fundamental concepts of relations. There are few important terms related to relational data model. Following description focuses on connecting mathematical model with practical design:

- **attribute** is an element which mathematical relation is composed of. In case of databases, it is column of a table.
- **domain** is a set of values, which attribute (column) can have. In real database, this set is reduced to data type of a column.
- **relation** (as in mathematical relation) is defined by particular value in each attribute. Of course the value has to be within an attribute's domain. From these values a tuplet is created. The relation is defined by all defined tuples. These tuples are represented by rows in a database table. Hence the relation in this case is defined by all rows the table contains.

As for integrity constrains, section 4.2.2 describes some general rules about designing the database and it also contains more specific description of integrity constraints.

Based on previous paragraph, it is possible to put together the basic idea of how the database looks like. Just to complete the image: the database is composed of a number of tables. Each of these tables has its description, which gives an order to the stored data. Tables can have integrity constrains defined. These can for example define that unique value is required in a column for every record in it. Through these integrity constraints, tables can be bounded together. Another part of relational database are stored procedures and functions. These are subroutines stored on a database server and compiled to run as fast as possible. Their purpose is to transfer some procedures to a server to minimize data transfer between the database and a client. Then we have triggers – these are basically stored routines, each one being associated with a table and write action on that table. The trigger is (as its name suggests) triggered once the associated action takes place on associated table. The last important element of relational databases is a view. It is virtual table which contains data gathered by (usually very complex) SQL query from other tables in the database.

There are some extensions of classical relational databases. For RPM a temporal database would be worth considering. It has built-in time aspect, which could be quite useful for example for logging and maybe for system debugging. However this particular extension would bring more cons than pros. The major one would be the incredible size of either a long or often used database. For that reason we will stick with classical relational model.

In this sections some basic concepts of relational databases were discussed. Now it is possible to connect these concepts with RPM and potential usage of these databases as a backend for RPM database. The first part of every database model is definition of its logical structure. When using relational database, this definition is always a part of the database. That's the first major advantage over databases like Berkeley DB – it is possible to see what data are stored in the database and what is their meaning. It also forces a bit of good design, because it is impossible to put different data to a column for every row. This can be overridden to some degree, but only the most straightforward design approach is considered now. The second part of a database model is definition of integrity constraints. When designed well, these constraints align the data with their semantics. They can also reveal and prevent errors which occur when the database is in use (for example primary key can prevent two versions of one package to be installed simultaneously). And the third part of a database model is a formal definition of a query language used by the database. Such language for relational databases is structured query language, or SQL. This language would bring several huge advantages to RPM:

- **lucidity:** when a new developer joins the project, he doesn't have to study the code extensively and there is a less need of doing documentation, because the SQL is pretty self-explaining, especially when the database design (or rather choosing of names used in the database) is good

- **encapsulation:** SQL is a declarative language. It means that we only tell the database which data we want to know and the engine does the rest. Hence the knowledge of how the data are stored on the disk or how to process them is not necessary.

- **portability:** it would be possible to write RPM library, which would not be depen-

dent on engine beneath, that means it would be possible to change engine version or even the whole engine without the library knowing about it

- **independence:** in current context it is meant rpmdb-rpm independence. When releasing a new version of RPM, it would be possible to change the database format by a set of `ALTER TABLE` commands. That is very important, because the database has to be extensible to allow implementation of future features.

Now when relational databases and their potential advantages in context of using them for rpmdb were listed, it is time to look at good design practices.

4.2.2 Database design

To design a database based on relational data model, it is first important to know that only scalar values are allowed. It is of course possible to override this rule to a certain degree, but it is against rules of a good design, because it takes away advantages one can get by using relational database.

Now when we know the basic concepts of a database, it is possible to design it. The best practice how to design relational database is to create its model first. To do this, Entity Relationship diagrams (provided by UML) are the most used approach. How to do the diagram isn't main focus of this thesis and neither is how to design database of those models. So let's skip to the description of how good designed database should look like.

Section 4.2.1 mentioned integrity constraints. Now something about what they are. Integrity constraints is a set of rules relational database can have which define what format and dependencies must be met for the data to be accepted by the database. There are two basic kinds:

1. **general:** these are valid for every relational database, no matter what data it represents. They are derived from basic concepts of relational data model.

2. **specific:** this group contains a specific set of constraints. They are database dependent. To be more precise, they are dependent on semantics of the data in the database.

From a point of view of a database designer we have different groups:

- constraints within the table (primary key, not-null column, ...)

- constraints defined between tables (foreign keys)

For following text a glossary is needed:

key attribute is attribute (table column) which is a part of a candidate key

non-key attribute is every attribute which is not key attribute

candidate key is a column (or a set of columns) for which 2 rules apply: value of this column has to be unique for every row present in the table and the key has to be non-reducible (we can't take a part of it away without loosing the uniqueness). It is possible to have multiple candidate keys per table, some of them may even overlap.

primary key is a special case of candidate key. Usually the simplest CK is chosen. There is only one PK allowed per table and there mustn't be (partially or fully) undefined value in it.

functional dependency an attribute is functionally dependent on another attribute when it has always the same value for one value of the attribute it depends on (for example a city can be functionally dependent on an airport code – for airport code, there is always the same city associated)

full functional dependency is usually defined in a context of a composite candidate key. When there is attribute functionally dependent on a composite key, fully dependency means, that it is dependent on all parts of the key combined, not only on some of them.

non-trivial functional dependency trivial functional dependency is defined in context of composite candidate keys. Each attribute of this key is functionally dependent on the key it is part of. Non-trivial dependency is every dependency when attribute is out of the CK it depends on.

transitional functional dependency is derived from transitional relation in mathematics. Attribute A is transitionally dependent on attribute C when there is attribute B exists, which is functionally dependent on C and A depends on B.

Once we have a database design, process called *normalization* takes place. This process defines rules and methods to transform design of each table to a good design. Good design then improves the subsequent work with data. Normalization has several levels. After achieving each level, we can say, that the table is in n^{th} normal form (NF).

1st NF is achieved once the table contains only atomic values. That means that there is no value which could be semantically split. For example address field. If we don't care about its values and we only need it as a text filled in order, it's fine to have one field called `address`. But if we need to work with components of the address (let's say with postal code), we have to divide address to several separate fields.

2nd NF has a prerequisite that 1st NF has to be achieved. Then a full functional dependency of every non-key attribute on every CK is required.

3rd NF has a prerequisite of 2nd NF achieved. The second condition is that no non-key attribute is transitionally dependent on one of CKs.

BCNF or Boyce-Codd normal form is a minimal level usually targeted to be achieved. It requires 3rd NF to be in place. Then it requires that all non-trivially dependent attributes are dependent on superkeys (basically this can be reduced to a statement that there shouldn't be a functional dependency between CKs)

4th NF requires BCNF to be achieved. Then the table must describe only causal relationship (i.e. one fact)

5th NF requires 4th NF. To achieve 5th NF, the table can't be without loosing the data.

4.2.3 Embedded SQL systems

There are two major categories of relational database systems – database servers and embedded libraries. We will focus only on the second one, although database servers have indeed some advantages as well. They handle concurrent access pretty well and they are still running which means potential performance boost, since there is no need to load the

data repeatedly from HDD. On the other hand, they are too large to be carried as an RPM dependency. Also the second mentioned advantage has its counterweight and that is memory consumption, which we would like to keep as low as possible.

SQL (Structured Query Language) is a common language supported by most relational databases. It is declarative language and it is specifically designed for relational databases. It supports all necessary parts which are needed: data creation, manipulation and controlling. It is defined by several standards – from *SQL-86* to *SQL:2008*. We will occupy ourselves only by engines supporting this language, because it is well known and can support some very handy features (e.g. simple database update between different versions of RPM).

The most widely used representative of embedded SQL systems is SQLite, but some other emerged in last few years. One of them is embedded MySQL and derived products (Drizzle, MariaDB). We will compare this relatively new embedded library with SQLite as current leader in this category.

First we should start at general specifications. Both solutions are embedded libraries, dynamically linked with the application code. SQLite is very widely used, which gives it solid and thoroughly tested code base. MySQL is even more widely used, but as a standalone server. Nevertheless the code base is the same for both versions of MySQL – standalone and embedded [11, 9]. Both databases provide programmer access through external tool, as shown below. So both databases should be equal from this point of view. The only real advantage here has MySQL and that is much better developer documentation. Also it is possible to run SQL server over the database and connect to it from external programs. That could be useful e.g. for debugging through user friendly interface.

```
# sqlite access to its database
sqlite3 <database-file>
# mysql access to its database
/usr/libexec/mysqld --default-storage-engine=MyISAM --datadir=‘pwd‘/data \
  --socket=‘pwd‘/sock --skip-grant-tables --skip-innodb > /dev/null 2>&1 &
mysql --protocol=SOCKET -S ‘pwd‘/sock
```

A brief description of databases' features should be introduced. MySQL offers several database engines with *MyISAM* and *InnoDB* as the most significant ones. MyISAM is based on the original storage engine used by MySQL (ISAM) and is still default storage engine today. It stores each database in separate directory and every table in the database is represented by 3 files (definition of table metadata, data themselves and index file). Utilities designed to check, fix and compress tables exist. From its other features, the most important is support for fulltext search. MyISAM is also much more faster than InnoDB, because it omits some operations bringing ACID compliance, such as transaction log. The only recovery-related mechanism is a flag, that table file was closed correctly. Although it can be used to detect some crashes, it surely can't detect all of them, which means a substituting mechanism has to be implemented. It is possible to provide parameter `--flush` to programs using MyISAM. In that case, stored data are synced to disc after each query is completed. This brings a good deal of robustness, but it slows down the whole operation up to 70 times (comparison test using 100 000 insert queries was performed). In further tests this feature will be turned off and a compromise solution will be presented in chapter 6.

InnoDB is much more robust and supports many more features, like foreign keys and triggers. It stores all databases (and of course all tables) in one file, called ibdata. Another

two files are used as transaction log (ib_logfile0 and ib_logfile1). Despite many features InnoDB has, there are some which are missing and are essential for good database design. It doesn't support fulltext search and in embedded mode it also doesn't support concurrent connections[11]. Because of these properties and its superior speed, MyISAM will be used further in comparison.

SQLite implements database engine, which is ACID compliant by default, but this feature can be turned off to increase performance. It supports much less features than MySQL[1], but some of the missing ones can be found as modules (e.g. the module handling fulltext search). However there are situations when one can find missing features would be very handy.

Some performance tests are needed to compare SQLite and MySQL. All further described tests were run on a testing database table `package`, taken from database described in section 5.2. For a complete description of MySQL version of this table, see section 5.2. SQLite version has only three modifications. Fulltext index is created by *FTS module* as described in [8], the index on `state` column is omitted (SQLite doesn't support them) and all columns with `enum` data type were switched to `varchar` (SQLite doesn't support enumeration data type). Test were also run both with and without fulltext index (FTS extension respectively). Again absolute numbers are given as a results of these tests. The reference machine was the same as it is section 3.2. Only this time Fedora 12 was installed (upgrade from the previous system, so it shouldn't have any impact on machine performance).

The first test parsed a file with a records for all 19119 packages in F12 repository. Records were stored in a simple file, which is attached on DVD, as is the script that generated it. Parsing itself wasn't measured and is (except for very tiny nuances) the same for both backends. When the whole file was parsed and loaded to memory, inserting itself begins. In both cases, it uses pre-parsed query and then just adds parameters to it. Details can be seen in the program on attached DVD. The program was run 100 times for each backend, no other processes significantly influencing the disk were started. Before program was run, the database has been initialized, disk caches have been dropped and after it was run, the database was deleted. These operations weren't subjects of interest, only the insertion itself was measured. Table 4.1 shows results of this test. Optimized measurements measured only the insert operation, non-optimized tests included opening/closing the database as well. Indexed measurement was taken on the table with respectively *fulltext index* or *FTS module activated*. As it is possible to see, optimized and non-optimized insertions took almost the same time and the difference between them is within statistical error. What is significant is the difference between time of insertion into indexed and non-indexed tables. Although the difference is this big for a large number of packages, we should think about every fulltext index used in the database, because it significantly slows down writing operations. And as for the original comparison, MySQL is quite better here, as one can see in the table.

	MySQL		SQLite	
	non-indexed	indexed	non-indexed	indexed
optimized	1.258 s	4.005 s	2.643 s	5.523 s
non-optimized	1.262 s	3.985 s	2.643 s	5.52 s

Table 4.1: Insertion of 19119 records to database

The second test focused on querying the database that was created by programs and scripts in previous test case. The measurement method was the same as it was for BDB in section 3.2. The same test was taken for both MySQL and SQLite. It showed that MySQL has significantly better results when the database is in the disk cache. Queries performed on the MySQL column without index were twice as fast and those running on MySQL column with index were three times as fast as for the same SQLite column respectively with or without index. The result were in favor for SQLite in case of performing query on the database which isn't cached and has fulltext index. The difference here was almost tenfold. Table 4.2 provides complete picture of this test's results.

		MySQL		SQLite	
		non-indexed	indexed	non-indexed	indexed
No cache	non-optimized	0.0542 s	0.4939 s	0.0694 s	0.0681 s
	optimized	0.0528 s	0.5477 s	0.0691 s	0.0679 s
With cache	non-optimized	0.0383 s	0.0212 s	0.0694 s	0.068 s
	optimized	0.0382 s	0.0212 s	0.0692 s	0.068 s

Table 4.2: Times of queries for random packages

That was performance. Now something about other database features. The size of all loaded data was in both cases 5 801 270 *bytes*. Four states of database were measured for each database. Table 4.3 shows the results. Comparison with BDB overhead won't be given at this time, because this database isn't representing full RPM database, as it was the case in section 2.6.

Besides database size and overhead, its memory consumption was measured as well. SQLite came better from this test. It was performed by executing query on the same database as before. This query looked up all packages having *rpm* in their name and the memory consumption was measured in several randomly chosen times after database query was performed (to better catch this information, some sleep calls are invoked in the program). In all moments, memory consumption was say about the same, with only small fluctuations. SQLite had significantly lower memory consumption – approximately 3.5 MiB (13.1 MiB in total with shared libraries, etc.), whereas MySQL took approximately 13.5 MiB (170 MiB total). That means SQLite is more convenient for systems with less memory.

		Database size		Database overhead	
		MySQL	SQLite	MySQL	SQLite
Without FTS	Empty	11292 B	4096 B	11292 B	4096 B
	Full	7069.08 kiB	8671 kiB	1403.78 kiB	3005.7 kiB
With FTS	Empty	11292 B	7168 B	11292 B	7168 B
	Full	13479,08 kiB	13068 kiB	7813.78 kiB	7402.7 kiB

Table 4.3: Database size and overhead comparison

Based on previous comparison, we will choose MySQL as a database which we will continue with. There are two main reasons: MySQL has larger feature set, which will be useful when designing the new database and it is significantly faster.

The speed of MySQL brings its disadvantages and that is memory consumption and disk space consumption. From table 4.3 we know that neither the overhead, nor the size of the database is that different from SQLite, so let's focus on the memory consumption. As it was stated previously, MySQL with all libraries consumes approximately 170 MiB, which can be considered pretty much. But it is very similar to value which one can get by running `rpm -qa` or for example `rpm -q rpm` and see its memory consumption (the appropriate command is shown below). Considering this fact memory consumption can be ignored, thus using MySQL doesn't have any significant downside.

```
rpm -q rpm > /dev/null & ps aux | grep rpm | grep -v grep |
  awk '{print $5 " kiB" }'
```

Chapter 5

Proposed format description

First of all, it is important to introduce a new designation of *"new RPM database"*. Although is pretty straightforward, a simpler one will be used sometimes. For the new design based on MySQL, designation *rpmdb-mysql* will be used.

One of the main goals of the new design should be reducing the amount of data transferred from database to RPM. This is automatically achieved by SQL concept defining that all relation attributes have to be atomic. However there is a large amount of RPM tags possibly contained within the package. It is important to consider that every single one of them might be in use in some programs or scripts. Because of this a new concept will be introduced in rpmdb-mysql. There is a certain amount of data, which every package must have and which are important to nearly every package and on every system (e.g. dependency-related data). These data will be stored in MySQL tables. All other data which are stored now and data which might need to be stored later as well will be saved in a different format. There are two possible approaches. Either it can be stored in raw format which is basically the same as rpm header format or it is possible to look for inspiration in different packaging managers and store the data in plain text. The first approach gives one advantage and that is code base. A code for loading packages can be used to load package information from the database as well. There is also disadvantage in a form of performance overhead – there are some parts of the code base which have to ensure package portability. The second approach gives us human readability and potentially quick access. All that is needed is to mmap the file to a memory and iterate through its content. Its content will be the same as it is in files used by dpkg:

```
Package: xserver-xorg-input-vmmouse
Architecture: amd64
Version: 12.6.4
Replaces: mdetect (< 0.5.2.2), xserver-xorg (< 6.8.2-35)
Provides: xserver-xorg-input-4
Depends: libc6 (>= 2.7), xserver-xorg-core (>= 2:1.6.2)
Description: X.Org X server -- VMMouse input driver to use with VMWare
This package provides the driver for the X11 vmmouse input device.
```

Of course it is convenient to include this new format in RPM files (possibly in a compressed form), so the code base could be the same for packages and database. That would mean that the first approach would loose its advantage and could be made obsolete entirely. Because this is not likely to happen over night, the following text will describe some specifics of plain

text data files, but let's consider this only as a vision which is not going to be implemented in near future. Instead, an implementation using RPM data headers is considered as primary. Thus text describing how text information is stored also describes how binary rpm headers can be stored. In both cases this division grants availability of all data and fast access to the most important parts of them. In other words, when implemented right, it will implement data cache in a form of SQL database. The most important parts stated above are the commonly searched data and common search and order parameters. In case some other information is required, a filesystem storage will be involved (query will fall back to reading whole headers from disk).

The code block above is just an example – actual names used in this file would have to be adjusted to RPM tags. There are three possibilities how to place all the packages:

- All packages in one file. This concept is used by dpkg. Package records are separated by a blank line (for binary form no separator is necessary, we always know header size). Thus a blank lines used for example in package description have to be modified for example by containing a single dot. This approach could have some performance issues when the file (10+ times larger than in Debian) becomes extremely large. A possible reason for this will be explained later. On the other hand this approach becomes handy during sequential access, which takes place either during `rpm -qa` or e.g. database rebuild.

- Directory structure. This approach is inspired by ports-based systems. It is the exact opposite of previous case – it is worse in sequential access (opening and closing many files), but its performance is much better in random access.

- Combined approach. This option combines advantages of both previous approaches. All the data are stored in a single file for great sequential performance and there is filesystem structure which serves as an index. This can be implemented either as standalone text files in directories or there can be only reference files containing the index to the large file (in order to avoid redundancy) in a form of number giving offset from the beginning of file. This number can be also stored in the database.

The first and the third approaches use large files to store complete data information. It is true that this is convenient for sequential access, but there are two big reasons why not to implement this. One of them is memory. The file containing headers can grow to enormous sizes and at that point it will cause more troubles than it will solve (mmapping can become too demanding for memory). The second reason not to implement this is deleting / updating information. Updating information in the header file will be difficult either way, but deleting will be much simpler in case of directory tree. That's why the rest of the database will be counting with this design.

Inspired by approach of some other packaging systems, the database is intended to be used by both RPM and its frontends like yum, hopefully minimizing redundancy of locally stored data. This is another reason for usage of plain text file – they are natural for higher scripting languages like Python, which *yum* is written in. Because of usage of *yum*-like tools, the database will contain more information than in current solution. There will be features like all installed and available packages stored in the database, current package state and repository which the package comes from.

There is one more aspect which should be described here and it is a lock file. As it was stated before, MyISAM engine doesn't support ACID transactions. Even if it did, this lock would have to be implemented, it just could be in the database. The point of this

lock file is that it will be created before the operation starts and it will be deleted after the operation safely ends. That means when rpm is started and this file exists, it is telling to RPM that the original transaction should be finished before the new one is started. To be more implementation-specific: if the lock file was implemented by a database table, it would have to be opened with `--flush` parameter which would sync the updated table to disk right away. If it is implemented by a lock file, `fsync()` has to be called right after the locking value is written.

5.1 Data model of SQL part

Entity relationship diagram 5.1 shows, how the SQL database part of stored data will look. Data type `byte` represents enumeration data type.

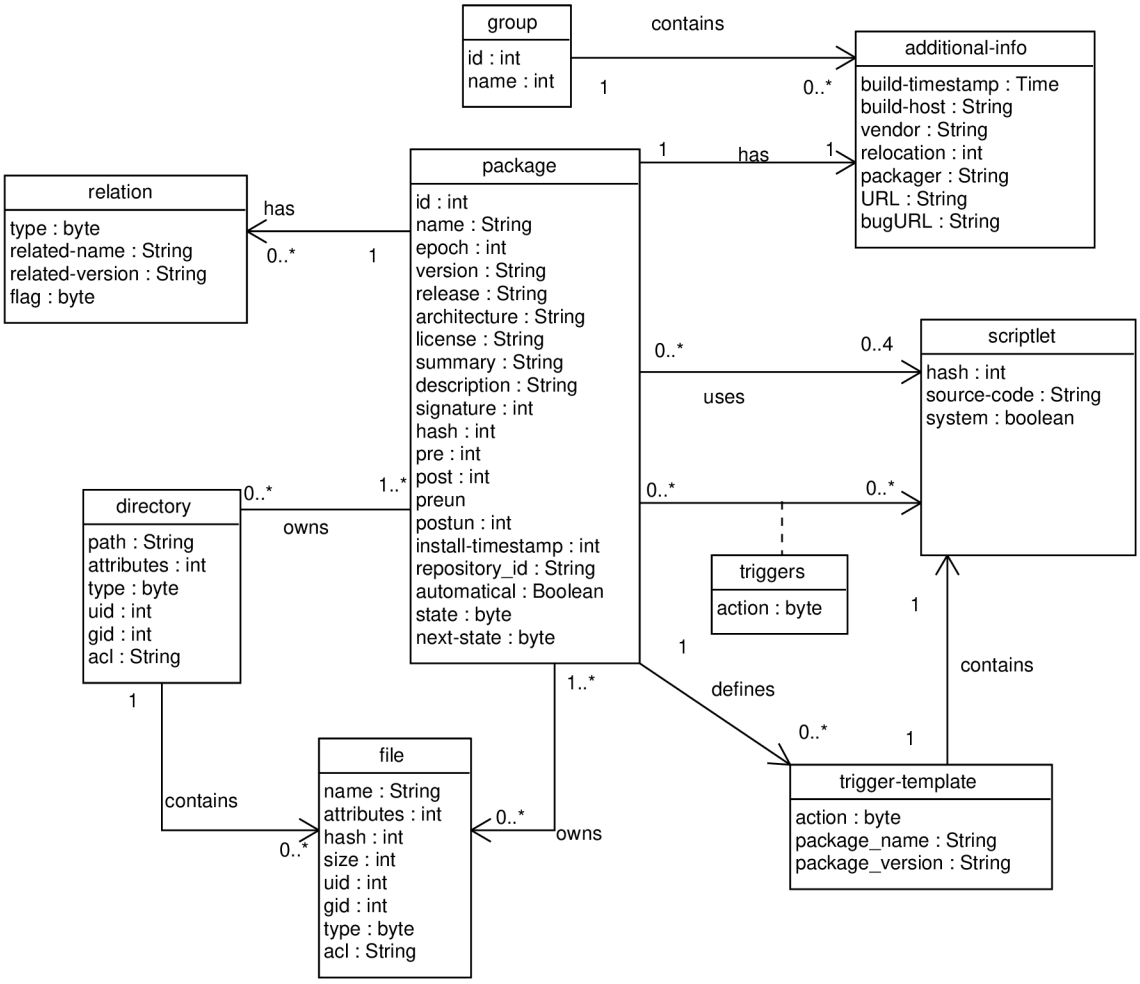


Figure 5.1: Concept of rpmdb-mysql

All package relations (dependencies, conflicts, provided features, ...) are represented by class *relation*. The *type* attribute defines which kind of relation this is. The *flag* has the same function as currently used tag `RPMTAG_REQUIREFLAGS` and similar ones have, hence it defines mathematical relation (less than, more than, ...) to border package version. Border

package version is then given by attribute *related-version*. As it is now obvious *related-name* gives the name of package, which is in relation. This concept might be considered even for feature-based dependencies.

Package entity itself contains the mostly used information for each package. This information can be further divided based on estimated (or later measured as well) frequency the data are accessed. The ones which are likely to be accessed more often are placed in entity package. The rest of them is placed in additional info. What data are stored where isn't important at this moment and so it isn't important which data are even stored and which aren't (this can be considered just as a proof-of-concept design and after all the data required to be stored can change over time). The division into two distinct entities (and subsequently into two different tables) is in place for potential performance boost. The two tables are in 1:1 relation. If some test are performed and if it turns out that it has no effect on performance, it is possible to join them in one. Currently they are conceptually divided because reading smaller tables should be faster (less disk blocks have to be loaded to memory).

There are some significant attributes of *package* entity. First of all, there is an *id* attribute, which might look like redundant, because every package is identified by its NEVRA. This attribute is present to simplify identification of related package in all other entities. Bottom line is that it will lead to less data stored on the disk, which means it in fact reduces redundancy. Another attribute which might seem redundant is package description, one can think about package summary as well. Because the database is meant as an index, it is convenient to keep stored data at the lowest possible level. By this logic, at least description should be excluded. The reason to include it is a fulltext search index that is defined on it. It is possible to search package based on what does it do – that is particularly useful for yum-like tools. Another interesting attributes are *state*, *next-state*, *automatic* and *repository_id*. These are meant as a support columns for yum and related tools, as it was described in chapter 5. The *repository_id* column is primarily intended to be equal to identifier of repository used by yum in its configuration files (enclosed in square brackets). The rest will be described in section 5.2. *Color* is special attribute, which is currently used as a set of flags. By setting specific bits of this color we can tell RPM that the package is for example multilib package. If no flags are set, this attribute is zero. Another significant attributes are present to outline how regular scriptlets will be stored. As it was stated in section 2.3, the package can have up to four scriptlets – `%pre`, `%post`, `%preun`, `%postun`. Triggers are stored separately. But first an introduction of the scriptlet concept:

The concept is based on *conary* and its tags. The idea of tags is good and modern, but not entirely sufficient. This concept takes the best of tags and complements it with some features that are required for distributions like Fedora. There will be some standard scriptlets provided by RPM, but each package can also carry its own scriptlets which will be then included in scriptlet database. The idea if this is simple: scriptlet can be reduced to its hash, which will be used as ID. If the package contains scriptlet which is already present in the database, it will not add it once more, but it will create link to the existing one. This concept is again designed to reduce redundancy.

There are three relations between package and scriptlet. First of them is *contains* relation. This is not displayed at diagram 5.1, explanation will be given below. This relation defines whether the scriptlet came with the package. Together with *system* attribute it forms an information when the scriptlet should be removed. When it is a system scriptlet, it won't be removed until the whole RPM is erased. Otherwise it will be removed once all packages owning the scriptlet are removed from the database. The information about own-

ership won't be stored for system scriptlets, because it is irrelevant for them. Information about ownership for other scriptlets is implied by other two relations between package and scriptlet. Once there is one of these relations present, it automatically implies, that the package owns given scriptlet.

The second and third relations define when the scriptlet will be used. The *uses* relation will be implemented as it is outlined in *package* entity. There will be four attributes storing information which scriptlet will be used at what time (empty value means no scriptlet is scheduled for that time). Maximal cardinality on the scriptlet's end is four, even though there seems to be no reason to store `%pre` scriptlet. It is present for the sake of robustness – an anticipation is that during installation the database will contain information about installation's process and in case of a system crash during installation and subsequent transaction finishing, an information about all the scriptlets might be needed (depending on a stage, when crash occurs).

There is one issue which is to be solved by rpm implementation and it is scriptlet tracking. The operation can fail in the middle of executing scriptlet. There are two possible approaches after such crash. We can either revert effects of the scriptlet or we can continue from the point of crash. The first approach is very complex, since a revert scriptlet would be needed to complement every classical scriptlet and even if this premise was fulfilled, some operations are not revertible by their definition. Certainly there are ways to solve this, but these aren't a subject of this thesis. The second approach is a little bit simpler, but again, there are many issues to be solved before implementing it – for example database can be designed to store a marker where the scriptlet ended, but to use it, rpm would have to implement a way to find it out first, not to mention a way to store scriptlet state (variables, etc.) would be needed. That's why these issues are only outlined here and they are not a part of the database design. Basically a shell interpreter has to be implemented before a database solving this part is designed.

The third relation stores an information about when the scriptlet will be triggered by some action on the package. Triggers are the same scriptlets as all others, but they are handled differently than regular ones, because their relation with packages is a little bit different. From database point of view, a trigger is a link connecting two packages in a similar way relation connects two packages. The only difference is that from the trigger point of view, both linked packages are equal in the relation – trigger is executed (therefore a lookup of a scriptlet is needed) once given action is performed on either one of these linked packages. More detailed description of trigger design will be provided in section 5.2.

The last important part of the design is a part storing information about files and directories owned by packages. The concept operates with the fact, that one file can be owned by multiple packages. In order to avoid database overload, only files of installed packages will be in the database. This could mean that package-file association is defined as 1:N. However, it is possible that two packages will claim the same file – even with previous limitation this situation means that package-file association is defined as *M:N* and the database has to be designed accordingly. Also in case of conflicting files from different packages, RPM should handle it – administrator will be either asked for solution, or the operation will fail entirely. One exception is possible and that is if files are equivalent – in that case file will be loaded to database and won't be overwritten. Equivalency means that files will have the same attributes and content. Since comparison of a file content can be non-trivial operation, there is an attribute *hash* in the database. Its meaning is to contain e.g. SHA-256 has of file contents (calculated at build time). In both conflict cases (equivalent files, one file replaces another) there should be an information kept about this

conflict in each file record. That's the goal of attribute *conflict*, which will be described in detail in section 5.2.

Similar concept applies for directories, except their content won't be considered. It will be possible for multiple packages to own one directory and the only condition is that its attributes equals for both packages. As same as before, attribute conflict determines which record is valid and which is not.

The very last entity in diagram 5.1 is group. This is again related with usage by RPM frontends, which are aware of groups. This entity has only informational character without any deeper meaning. This entity might be also modeled as an attribute of package or additional info. Considering that the database engine probably stores the string describing each string separately, this is better way to reduce size of a table and thus increase performance (less blocks might be needed to be read from HDD), because only one number is used for representation of a group in case this information is not target of a lookup.

5.2 Final design

Now follows SQL code used to create the database equal to diagram 5.1:

```
CREATE TABLE IF NOT EXISTS `package` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(256) NOT NULL,
  `epoch` tinyint(3) unsigned NOT NULL,
  `version` varchar(16) NOT NULL,
  `release` varchar(32) NOT NULL,
  `arch` varchar(8) NOT NULL,
  `os` varchar(16) DEFAULT NULL,
  `color` int(10) DEFAULT NULL,
  `license` varchar(32) NOT NULL,
  `summary` varchar(256) NOT NULL,
  `description` text NOT NULL,
  `pre` mediumint(8) unsigned DEFAULT NULL,
  `post` mediumint(8) unsigned DEFAULT NULL,
  `preun` mediumint(8) unsigned DEFAULT NULL,
  `postun` mediumint(8) unsigned DEFAULT NULL,
  `install-timestamp` int(10) unsigned DEFAULT NULL,
  `repository` varchar(128) DEFAULT NULL,
  `state` enum('purged','pre','files','post','installed') NOT NULL,
  `next-state` enum('install','update','remove') NOT NULL,
  `automatical` enum('0','1') NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
  UNIQUE KEY `name_2` (`name`,`epoch`,`version`,`release`,`arch`),
  FULLTEXT KEY `name` (`name`,`summary`,`description`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1;
```

Id was chosen here as a primary key, because it is the smallest candidate key. Seeing *epoch* column, one can think that many packages don't have any epoch stated, so it should be possible to store `NULL` there. In fact from the definition it is not possible – this column is used as a part of an identification of the package. In SQL terminology, is is a part of *compound candidate key*, which means it cannot be `NULL`. Because of this, the default value

(simulating NULL) is zero. To sum up – if the package doesn't have any epoch stated, it will be stored as zero. In this case, the epoch number shouldn't be used nor displayed to user. Size of all text fields was chosen, so the common string used in that particular field can be filled in. *Pre*, *preun*, *post* and *postun* fields and their data types correspond with scriptlet identifier. *Repository* can be NULL, because the package can be installed from hard drive. Similar rule applies for *install-timestamp* – the reason is that package might not be installed. State here defines the state package is currently in – it is either installed or uninstalled. The other states signify, that package has already gone through that given stage during installation/uninstallation/upgrade. Next state is derived from concept used in Debian-based distributions. It is a marker of the direction we are planning to follow with the package. It can be particularly useful when transaction with the package fails in the middle – in combination with current state we can easily see where did it end and how is it supposed to continue. Automatic flag just states that the package has been installed as a dependency and once every package depending on it is removed, it can be safely removed as well.

```
CREATE TABLE IF NOT EXISTS ‘group‘ (
  ‘id‘ smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  ‘name‘ varchar(128) NOT NULL,
  PRIMARY KEY (‘id‘),
  UNIQUE KEY ‘name‘ (‘name‘)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

Here it is possible to have index on both column, because we can safely assume that this table won't have many records in it – that means the index won't grow too large (which would mean major slowdown).

```
CREATE TABLE IF NOT EXISTS ‘additional-info‘ (
  ‘package‘ int(10) unsigned NOT NULL,
  ‘group‘ smallint(5) unsigned NOT NULL,
  ‘build-timestamp‘ int(10) unsigned DEFAULT NULL,
  ‘build-host‘ varchar(64) DEFAULT NULL,
  ‘vendor‘ varchar(64) DEFAULT NULL,
  ‘packager‘ varchar(128) DEFAULT NULL,
  ‘url‘ varchar(128) DEFAULT NULL,
  ‘bugurl‘ varchar(128) DEFAULT NULL,
  ‘signature‘ text,
  PRIMARY KEY (‘package‘),
  KEY ‘group‘ (‘group‘)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

This table contains mostly non-essential (but potentially often accessed) information for the package, that's why majority of columns can have NULL values. It is linked with previous table through column *package*, which represents the package id here. Column *group* is indexed, because an application may want to look up all packages belonging to one group.

```
CREATE TABLE IF NOT EXISTS `directory` (
  `path` varchar(996) NOT NULL,
  `package` int(10) unsigned NOT NULL,
  `attributes` smallint(5) unsigned DEFAULT NULL,
  `acl` varchar(256) NOT NULL,
  `user` varchar(16) DEFAULT NULL,
  `group` varchar(16) DEFAULT NULL,
  `flags` set('config','doc','donotuse','config-missingok',
  'config-noreplace','ghost','license','readme','exclude',
  'pubkey','policy') DEFAULT NULL,
  `conflict` enum('no','replaced','valid') NOT NULL,
  PRIMARY KEY (`path`,`package`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE IF NOT EXISTS `file` (
  `path` varchar(996) NOT NULL,
  `package` int(10) unsigned NOT NULL,
  `attributes` smallint(5) unsigned DEFAULT NULL,
  `acl` varchar(256) DEFAULT NULL,
  `user` varchar(16) DEFAULT NULL,
  `group` varchar(16) DEFAULT NULL,
  `hash` varchar(128) DEFAULT NULL,
  `size` bigint(20) unsigned NOT NULL,
  `flags` set('config','doc','donotuse','config-missingok',
  'config-noreplace','ghost','license','readme','exclude',
  'pubkey','policy') NOT NULL,
  `conflict` enum('no','replaced','valid') NOT NULL,
  PRIMARY KEY (`path`,`package`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

These two tables represent everything related to the filesystem. M:N relation between packages and directories and files respectively could be represented by tables like *file-owner* and *directory-owner*, but that would disable the option that one file can have different characteristics in two different packages. Hence this solution was designed. In section 5.1 a conflict attribute was outlined. Here it is specified as enumeration which might have values `no`, `replaced` or `valid`. The first one says that RPM is not aware of any conflicts with this file in the filesystem. Replaced means that there is a conflict and the original package file has been already replaced by another one. In such case, it would be nice if there was a backup managed by RPM. After package with conflicting file is erased, the original one can be restored. Columns `user` and `group` represent respective system entities. In diagram 5.1 they were denoted as UID and GID with int data type. Here they are text based because of internal package format. But theoretically they can be both numbers if converting algorithm is implemented. Attributes are stored in a form of four digit integer. Each digit will signify one digit of user access rights. *Flags* column in both cases can be extended on every flag that RPM provides. This is just a concept design. It is possible to notice that even though an association between file and directory exists in the model, it is not implemented here. This has two reason. The major one is that MyISAM has significant limitation – only 1000 bytes are allowed to be indexed in each row. That means a limitation for directory and file path length. In such case half length would be maximal and that's

not an option. In fact those 996 characters is a bold limitation, since filesystem usually allows larger. The second reason is that the association can be implemented by substring lookup directly in the code of RPM.

```
CREATE TABLE IF NOT EXISTS ‘relation‘ (
  ‘package‘ int(10) unsigned NOT NULL,
  ‘type‘ enum(’requires’,’recommends’,’suggests’,’provides’,’conflicts’,
              ’obsoletes’) NOT NULL,
  ‘related_name‘ varchar(256) NOT NULL,
  ‘related_version‘ varchar(16) DEFAULT NULL,
  ‘version_relation‘ enum(’lt’,’le’,’eq’,’ge’,’gt’) DEFAULT NULL,
  PRIMARY KEY (‘package‘,‘type‘,‘related_name‘)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

One thing worth of noticing in this table may come to an attention. Since there is only one table representing all relations, it can grow to enormous sizes. There are about twenty thousand packages in fedora everything repository. Now another five thousand packages in updates, five hundred in rpmfusion. This is commonly used Fedora repository configuration and it makes around 25 500 packages, which will be in the database simultaneously. Each of these packages has a number of relations, whether they are provides, requires or other. From section 2.1 we know that the mean number of features per package is twenty two, which means that the table will have at least half a million records. Considering that this table will be under heavy stress during dependency solving, the solution is needed. There are two solutions. Either to divide this table and for every relation type use its own database table or to consider change of relation concept. This change should replace current feature and file-based dependencies with package and virtual package dependencies. That should reduce size of this table. In fact virtual package dependencies can be considered as an equivalent of feature based dependencies. The only difference is a granularity, which is coarser in case of virtual packages. But before redesigning the database, this concept has to be tested if it even needs redesigning.

```
CREATE TABLE IF NOT EXISTS ‘scriptlet‘ (
  ‘hash‘ int(10) unsigned NOT NULL,
  ‘source‘ text NOT NULL,
  ‘system‘ enum(’0’,’1’) NOT NULL,
  PRIMARY KEY (‘hash‘)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE IF NOT EXISTS ‘trigger‘ (
  ‘scriptlet‘ int(10) unsigned NOT NULL,
  ‘owner‘ int(10) unsigned NOT NULL,
  ‘trig_name‘ varchar(256) NOT NULL,
  ‘trig_version‘ varchar(16) DEFAULT NULL,
  ‘trig_flag‘ enum(’lt’,’le’,’eq’,’ge’,’gt’) DEFAULT NULL,
  ‘action‘ enum(’post’,’preun’,’postun’) NOT NULL,
  PRIMARY KEY (‘scriptlet‘,‘owner‘,‘action‘)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

Finally, these two tables represent scriptlets and their usage by packages. As it was outlined in section 5.1, the ownership of scriptlet is implied by existence of a trigger or by particular

attribute in package entity. Hence there is only one extra attribute added to scriptlet and that if a flag saying whether scriptlet is owned by RPM system. The purpose of this was already described. Primary key of trigger table is compound and it has three items. It basically means that each scriptlet can be used by multiple packages as a trigger and it can be used by one packages as a trigger on multiple occasions. Possible occasions are only three, because current RPM doesn't allow scriptlet to be triggered before package installation, only afterwards. See section 2.3 for details.

Some of used data types and constructs are MySQL specific. First of all unsigned is not common in relational databases and its meaning is obvious. What might not be obvious is `auto_increment` column specification. This can be used only on numeric columns which are primary keys. It means that if no value is specified here while inserting new row a default will be picked as a sequential value of an internal counter. This value of course increases every time it is generated. It is possible to manually specify number in auto incremental column. One disadvantage is that it is very difficult to reset this counter. It is important to keep this on mind and give the column its range accordingly. In our case 32-bit number will be enough for quite some time. There are two data types that are not common in relational databases – `enum` and `set`. Both are text-based data types and their functionality is obvious from their names. Enumeration allows us to pick one of defined values and set allows us to pick multiple values out of defined ones. Parameters engine, charset and their values are self-explaining.

Section 4.2.2 described normal forms in relational data model. Let's now analyze database compliance with these normal forms:

- **Conditions of the first normal form are met**. The only columns which might be unclear are following: `package.color` is a set of flags, which can be defined as multiple columns with `bit` data type. But it can be also described by single column with `set` data type. Considering how RPM currently works with colors, this is equivalent and better representation. Thus it is atomic. Another two candidates for compound columns are `directory.acl` and `file.acl`. Those would be compound if we wanted to use their parts separately. But since we want to use them as a whole, it is atomic as well.

- Only columns looking like they might be breaking the second normal form are columns in file and directory tables. Here it is sufficient to remind that every package can have its own version of file (or directory) with different attributes. Hence full functional dependency is achieved. Another arguable column is `package.arch`. Here it is important to remember that even though in most cases attributes will be the same for different architectures (which would mean they wouldn't be fully functionally dependent on NEVRA), it is possible that package will have different attributes for different architectures (rpm spec files allow conditional processing base on architecture). That implies that full functional dependency is preserved. Hence **the second normal form is achieved**.

- **The third normal form is also achieved**, the database was originally designed to eliminate redundancy (i.e. functional dependency of non-key attributes)

- Boyce-Codd normal form can be broken in tables with multiple candidate keys in case they are functionally dependent. The only case this rule might seem broken is in table file, where `file.hash` might seem to be functionally dependent on compound primary key. This however is not the case, because hash depends only on file

content. This content always depends on package and file, but there is a situation which breaks functional dependency – the package owning given file can have different number. Thus another package owning the same file with different content would lead to different hash. Hence **all conditions of BCNF are met and the form is achieved**.

- higher normal forms weren't targeted by this design, so they won't be analyzed

Solution of the entire database part was introduced, now only disk storage part remains. Current database is placed in `/var/lib/rpm`. The new one will be placed there as well. Following text will consider every directory path as relative to this location, i.e. `/` will in fact be `/var/lib/rpm`. The database directory structure is very simple. Every package will have its "home" directory situated in

`/name/<name>/<nevra>/`

Where `<name>` represents package name (e.g. `rpm`) and `<nevra>` represents complete identification string (e.g. `rpm-4.8.0-14.fc13` or `aspell-en-50:6.0-11.fc12.x86\_64` – notice that the first package doesn't have epoch and arch). The complete path would then look like

`/name/rpm/rpm-4.8.0-14.fc13/`

Every home directory will have at least two files in it:

- **header.dat** will be rpm package header stored in file. Compared to entire package header, this won't have some unnecessary parts, such as RPM lead and magic number. It will simply be a header index and its payload.
- **header.txt** will be a textual representation of header file. It won't be compressed, because storage place isn't expensive and user typically demands speed of every operation he performs, including operations with this textual file. *This textual file is here just as a concept.* As it was stated before, it is good to include it in design plan, but the reality in near future will most likely lie with the header file.

Because package in database is represented by its numeric ID, the disk data part will include ID-based referenced to packages. They will be in a form of links, either hard or symbolic. Links will eliminate redundancy which would be present if copying was used. The directory structure will follow this model:

`/id/<id>/ -> /name/<name>/<nevra>`

Considering reader has knowledge about naming conventions explained previously in this section, the description is self-explaining (`<id>` is package id stored in the database). In this file-based database part only installed packages will be present, because of potentially enormous data volume that would have to be downloaded and stored. Essential information about every package available in repositories will be stored in the database. Sample directory structure is included on attached DVD.

Chapter 6

Design and implementation of new RPM database module

The goal of this thesis is to analyze current rpmdb API and if possible, provide its implementation for newly designed database, using MySQL as backend. If this won't be possible, the ideal goal is to implement entirely new API for the database. The purpose of this API should be to serve as a base for completely new implementation of RPM. But in case an old API can't be implemented, just to create sample programs demonstrating work with new database is a sufficient outcome.

6.1 Current API

Since the priority is to design API compatible with the current one, it is important to thoroughly analyze it and identify main points of contact which should be preserved in order to ensure the compatibility with the rest of system. Described API is a part of *rpm 4.7.1*. The rpmdb module of rpmlib has two different parts – *backend* and *rpmdb* itself. Backend part contains only the basic functions encapsulating the BDB function calls. These are adding only the very basic set of unrelated code, like setting important flags, handling errors and setting default values. Backend API corresponds with API of BDB. The whole idea behind this API was most likely possibility of switching backends and this API is supposed to be an interface between database and the rest of rpmlib. Despite good idea of this dividing API, it has one big disadvantage – because the original API has been designed to correspond with BDB API, it is difficult to adapt any different database to it. Embedded MySQL is a good example of it. BDB uses two different functions for reading and writing data, but MySQL uses only one multipurpose function. This situation perfectly illustrates the problem. Now its important to identify key functions of rpmdb API. They will be described in following list. Items in this list represent only those parts of current API, which should remain preserved. There are some other important parts, but those either are BDB specific or aren't compliant with the the definition of API term. Also some function which can be implemented as simple aliases are omitted. This is just a core part of API.

- **rpmdbAdd**: this function adds new package to the database (package header is expected as an input)
- **rpmdbRemove**: removes a package from the database, based of its ID number (how it gets it isn't important – this makes the whole process flexible)

- **rpmdbSync**: calling this makes the database store its data to disk, instead leaving this duty to operating system. Syncing behavior of current rpmdb will be very difficult to to emulate, because MyISAM engine simply doesn't have this user-forced sync capability.
- **rpmdbOpen**: opens the database for further operations. Because current solution consists of several databases, this function has to be called multiple times to open all of them.
- **rpmdbClose**: opposite of rpmdbOpen, this closes one database file. Multiple calls are required for closing all databases.
- **rpmdbInit**: initializes an empty database on specified directory
- **rpmdbVerify**: consistency check of rpmdb (sets the flag for BDB engine to verify the database before closing it)
- **rpmdbInitIterator**: this initializes match iterator structure. These match iterators will be closely described below
- **rpmdbAppendIterator**: adds another set (of package numbers) of packages to match iterator. Added package numbers aren't inspected for their existence in database. This is currently the only option how to create a match iterator without any lookup in secondary database. Let's say, we have a list of packages that have been installed during one transaction and we want to perform some operations on them. This is the way, how to assemble their list in form of match iterator.
- **rpmdbPruneIterator**: removes given packages from match iterator. This can be used for example as deleting the reference to deleted package in secondary indices
- **rpmdbNextIterator**: moves to the next item in match iterator and parses it
- **rpmdbGetIteratorOffset**: returns package ID (index number in Packages database)
- **rpmdbGetIteratorCount**: returns a number of packages which are selected by match iterator
- **rpmdbFreeIterator**: destroys structures associated with given iterator
- **rpmdbSetIteratorRE**: adds filtering rule to match iterator. Filtering rules will be described below
- **rpmdbRebuild**: rebuilds the database – this brings better performance and less overhead by ordering the data in database and optimizing indices in both primary and secondary databases

One very important principle was mentioned and that is the *match iterator*, implemented by structure *rpmdbMatchIterator* and its several handlers. This structure represents the means of searching in the database. There are two basic types of search performed on the database. The first of them is direct lookup of a package by its number and the second one is lookup using one of secondary indices. In both cases, the lookup itself is performed by function *rpmdbInitIterator*. This function fills in default values to rpmdbMatchIterator and then decides what to do depending on the database it should perform the lookup on.

If it's operating on *Packages* database, it only stores the number of package which should be looked up. If it operates on one of indices, it opens required index and looks up a set of packages which fit lookup criteria. This set of packages is then stored as lookup result.

The second part of this concept is implemented by function *rpmdbNextIterator* which iterates over the list of packages stored by rpmdbInitIterator. Each call of this function means it goes to the number of next package in the list, retrieves its header from Packages database and parses it. The header it then returned. If the end of matching packages list is reached, NULL is returned. This function also implements secondary lookup (or rather secondary selection). This subsists in utilizing *match iterator regular expressions*. These can define a series of conditions which the package we want has to meet. All of these conditions have to be met, which means they are connected with logical operator `AND`. This behavior strongly supports claim from section 3.1, that loading the whole header is a problem. Here another problem is presented – some headers are loaded only to be dropped directly afterwards.

In listed functions above, there is one particular that might seem to be missing and that is function handling database updates. It doesn't mean that current solution can do update only by deleting and re-inserting package. Of course there is a was to perform updates, but it is pretty obscure. To perform update a lookup is needed first. After match iterator is retrieved as a lookup result, an iteration through it is needed. If desired package header is found during this iteration, the modification is done directly in structure representing the header and a flag is set which indicates that header has to be updated in database (function `rpmdbSetIteratorModified`). Pointer to header structure is stored in match iterator and described modification is done in the original header, not on its copy. That implies that match iterator has in fact a pointer to updated data. In ordinary circumstances every time rpmdbNextIterator is called it drops the header from previous iteration and replaces it with the current one. But if the update flag is set, the header is transformed back to byte string and stored in the database again. This brings not just obscurity, but it also brings another slowdown – again we have to work with the entire header, not just with its part that has to be updated. Not to mention we have to iterate through headers to achieve an update.

After performing a deep analysis, it turns out **there is no acceptable way to implement new functionality into current interface**. There is one main blocker and several smaller issues. Let's begin with the smaller ones. A list of the key functions was already presented. During analysis a complete list of functions used outside of rpmdb was compiled. It showed that the rpmdb API was not well designed. To be more specific, it isn't prepared for another data format in the database. That means some functions (from those which are used from outside of rpmdb) would disappear entirely when using backend design other than the current one, simply because they would become needles, or because they'd be not applicable to another database paradigm. Usage of data structures specific to database paradigm might also cause complications. But the main blocker is the design of rpmlib (and subsequently whole RPM) itself. The main goal of new implementation was to avoid loading entire headers from the database. But this issue doesn't lie in rpmdb, but it is embedded in every part of RPM – every component was designed to work with the whole header and extract only needed information from it. This brings a simple and final answer whether to implement API compatible with the current one: *no*! It might be possible to adapt the new implementation: extract only needed information from database, fill it to header structure and return this structure incomplete. But this would mean either a new parameter or a set of functions should be added for this purpose. Also a calls of these functions would be needed in every place of rpmlib which is using rpmdb iterators.

6.2 Use cases analysis

As it has been explained in section 6.1, in case old API can't be implemented, either new API has to be built or it is possible to implement just some rpmdb use cases to demonstrate work with new database. In this thesis only some use cases will be implemented – design of new API would have an extent of entire bachelor or diploma thesis. Some thoughts about this will be described in section 6.4. Complete list of rpmdb use cases with code references and description what does referenced code do is described in attachment. Here just a brief list:

Find packages:

- find packages based on their NEVRA (either name, name+version, or the entire NEVRA)
- find packages which have one of specified IDs
- find packages which provide given feature or given basename
- find packages which require given feature
- find packages which conflict with given feature
- find scriptlets (or their owners) triggered by current operation
- find out how many packages are installed
- find packages in given group
- find package based on its hash
- find owner of given file
- find all files sharing given basename
- find files based on their fingerprint
- filtering lookup result set with various parameters (color, list of IDs)

Inspect given package:

- find out if package provides features it obsoletes
- retrieve package ID

Other actions:

- install package
- remove package

It would be good to implement at least one write and one read operation. From read operations mostly used is NEVRA lookup in various forms. As a write operation obviously at least installing a package is needed.

6.3 Use cases implementation

Sample programs attached on DVD were implemented to demonstrate the main advantages and disadvantages of new solution over the original one. This section offers a brief summary and description of the implementation.

Everything has been prepared in one package, which is ready to go, the only prerequisite is installed MySQL client, MySQL server (for init script) and MySQL embedded library. The package has one directory and several files. The directory is important, it will be root directory for the database. There are two init files – `init.sql` and `init.sh`. To initialize database, you can simply run `./init.sh`. The script utilizes the other mentioned file to create basic database schema. Files `common.c` and `common.h` contain mostly some routines, data structures and definitions related to the current implementation of RPM. And finally, there are source files for individual programs. It is important to know that these programs work with the database designed in previous sections, but they work with one simplification in the concept – only installed packages are in the database. To compile these programs, it is important to set environment variable specifying database home dir and afterwards run make.

```
export RPMDB_HOME=‘pwd‘/data
make
```

This will compile programs and prepare empty database. Some testing packages can be downloaded by `yumdownloader <package>`. These downloaded packages can be installed by calling `./install <path-to-package>`. Although it has been tested on some packages, the magnitude of RPM system suggests that it is possible to run to an error because there might be a package containing some combination of tags which has not been encountered and thus it might be problematic. However this can be acceptable for a proof-of-concept program like this.

Before describing the first program itself, some basic concepts of embedded MySQL should be explained, so the reader understands the code better. Every connection to the database is represented by structure `MYSQL`. This is a general descriptor, which has to be initialized, some options have to be set and finally the connection itself has to be made. Following code demonstrates a connection sequence (because it is only a demonstration, so the code is also demonstrative).

```
MYSQL mysql;
static char *server_args[] = {
"this_program",
"--datadir=/path/to/data/dir",
"--skip-innodb",
"--default-storage-engine=MyISAM",
"--skip-grant-tables" };
mysql_library_init(<arg-count>, server_args, NULL);
mysql_init(&mysql);
mysql_options(&mysql, <option-id>, <option-value>);
mysql_real_connect(&mysql,NULL,NULL,NULL,<database>,0,NULL,0);
```

Possible arguments are the same as arguments for MySQL daemon. Other options can be given to connection by `mysql_options()`. It is important to give at least and option to activate embedded library mode, otherwise it won't work. The counterpart of init sequence is very simple, just call `mysql_close(&mysql)` and `mysql_library_end()` afterwards.

The second important concept of MySQL are prepared statements. A big part of the code is related to prepared statements. The concept is to prepare a query with some places where particular values will be known later. Another reason is where one query should be used in iteration, only with different values each time. The code demonstrating prepared statements is following:

```
MYSQL mysql;
MYSQL_BIND package_bind[8];
MYSQL_STMT *package_statement;
char *package_query = "INSERT INTO `package` "
"SET `name` = ?, `epoch` = ?, `version` = ?, "
"    `release` = ?, `arch` = ?, `license` = ?, "
"    `summary` = ?, `description` = ?, `state` = 'installed', "
"    `automatical` = '0'";
#define PACKAGE_QUERY_LENGTH  196
fill_bind(package_bind);
package_statement = mysql_stmt_init(&mysql);
mysql_stmt_prepare(package_statement, package_query, PACKAGE_QUERY_LENGTH);
mysql_stmt_bind_param(package_statement, package_bind);
mysql_stmt_execute(package_statement);
mysql_stmt_close(package_statement);
```

Of course some other parts are added when considering retrieving data from prepared statements which represented `SELECT` query. These are not important right now. The code above shows that a lot of code is needed for simple query to be executed. The important part begins after definition of `PACKAGE_QUERY_LENGTH`. The first routine isn!t a part of MySQL, it is a placeholder for any routine preparing `MYSQL_BIND` structures. The principle is that every parameter which is to be given later is replaces by `?` and this placeholder is replaced by actual parameter in routine `mysql_bind_param()`. After binding the right parameters, it is possible to execute the query. Filling the structures representing parameters is more complex and it would be pointless to describe it here. Much better description is provided by the code itself. One more important thing about parameters. The query expects exactly as many parameters as there are question marks in the query. So it is common mistake to provide an array which is not large enough and end up with segfault. There are some situations in which this parameter-binding approach isn't desired. For such situations there is a simpler solution – a call of `mysql_query()` routine, which replaces practically all the routines in code above. The only thing it wants is query string and its length. The string isn't processed further, it is just given to the database engine.

Now when important MySQL concepts were explained, it is possible to describe the implementation itself. The code of the install program demonstrates one of the greatest challenges which has to be dealt with when designing a new database and implementing a new API on the top of it – and that is format conversion. The `main()` routine handles the majority of operations directly related with inserting data into the database. It uses some minor service functions like getId which returns an ID number of last inserted record to the database. This is possible for IDs which are automatically created

by MySQL in `AUTO_INCREMENT` columns. Most functions in the program are designed to transform RPM tags to MySQL bindings. Every bind has to be initialized, filled and checked before it is bonded before execution. Functions `parse_file_structure()`, `parse_relation_structure` and `parse_trigger_structure()` demonstrate how difficult is work with current package format. For example for files and directories, each of their attributes (mtime, access restrictions, link targets, ...) is stored in separate array. In case the file doesn't have this attribute filled, en empty value is presented, usually in a form of zero byte. Transforming this design into bindings must have two stages. In the first stage, the data are identified and stored in a temporary pointer variables. Then these variables are iterated over and correct values for each record are extracted and inserted into bindings. Then these bindings are used in query execution which takes place in `main()` function.

The `query` program is much more simpler and it shows how easy is extracting information from the database in comparison with current solution. A few use cases from section 6.2 have been implemented. Function `queryByLabel()` implements lookup based on NEVRA or any part of it as it is described in comments of [20]. That means at least a name of the package has to be given to the function. All other arguments are voluntary, but the basic concept has to be met – epoch, release and architecture can be only given when version is also present. There is also argument `mode`. This argument is meant for all lookup functions. Since a number of use cases were only about finding out either about existence or count of packages matching given criteria, a switch between these modes should be implemented. This is exactly the purpose of `mode` argument. This argument is also present in every other function in `query` program. Function `queryById` looks up which of given packages exist in the database by testing symlinks in filesystem part of the database. Functions `queryWhatRequires` and `queryRequires` demonstrate lookup of relations in both directions (we know package and want to know all of its relations vs. we know a relation and want to know which packages have it). There are some other functions implementing other lookups in the database. They all work on the same concept, they only use different queries, so there is no need to describe them here further.

The last program is the one which deletes package from the database. Some dependency checks and similar stuff are performed only as a demonstration, but full coverage of this isn't a part of this thesis. Of course if full database API was designed, this program would use it to perform those checks related to the database. The `delete` program is very straightforward – deletion from every single part of the database is handled by specific routine. All checks are performed in function `checkDeletion` or its subroutines. After everything is checked, the deletion is performed from both SQL database and the disk part.

6.4 Possible API design

This section should give only a very rough outline of what should new API support, and what might be the design needed to implement such functionality easily. First of all it's important to define functionality it should support based on use-case list from section 6.2. We will start with selection related functionality.

- support for simple selection
- support for multiple selecting parameters connected with logical operator AND. No other logical operations are needed by current solution. Although it would be better to think about them and at least prepare supporting data types for them.

- support for version comparison (to evaluate epoch-version-release string)
- support for comparing, arithmetical and logical operators. Logical operations will be necessary for filtering based on package color
- usage of comparing operators must not be limited to comparing field to a value, but also comparing two fields with each other is necessary. This is utilized in use case, when a goal is to find out, whether the package provides what it obsoletes.
- working with a set of packages in selection (this is used quite often - either to select those in a list or exclude them)

Now projection and sorting:

- retrieve one or more information from one table
- retrieve aggregated data. Currently only a count is needed. Some use cases of this aggregated value reduce this requirement only to an action to find out whether given records exist. Considering this in a design can bring a performance boost for this kind of queries, since there is no need for the database to walk through all records and simply first one will mean return from function.
- sorting result by one or more parameters. This is not really a requirement from use cases, but from observation, it will lead to cleaner code of the rest of rpmlib.

There are some unlisted, generic use case requirements. As same as in section 6.2, some service operation requirements such as opening and closing the database aren't listed.

As for the design outline: there are several approaches which might be chosen. The most simple way would be if entire rpmlib would be aware of underlying database and its structure. This is the case for other packaging managers, such as conary. The design of this approach basically defines only very basic API. Let's say the design goal it to provide at least some degree of freedom. Using SQL database it would be represented by possibility to switch backend engine without the need for modification of every code using database. Such code has to meet two conditions: first of all all queries have to be SQL compliant, no engine specific extensions used. The second condition is that backend API has to be written such as it is in current solution. This backend would follow generic principles used when working with relational databases (cursors, SELECT/INSERT/UPDATE/DELETE operation character, num_rows, num_cols, ...). An advantage of this concept is its simplicity – minimum amount of code is needed to implement rpmdb module itself. The main disadvantage is distribution of code – in case database schema changes, every part using database will have to be checked and adjusted.

Still simple, but more powerful approach is to use identified use cases and implement service function for each one of them. Attached list of use cases is verbose, so this should pose no problem. Usage of this approach however would lead to one of three possible outcomes: either the API would be bent in overlying layers, modification of API would be needed or new alias functions will arise. Now an explanation of all three cases. Everything starts with the need for database functionality which is not yet developed (or maybe only a close one exists, but it doesn't fully comply with what is needed in current one). Now there are three options. Programmer can use that currently existing solution and slightly modify its results so they are coherent with what is needed. Another possibility is that this modification is implemented in rpmdb module, which is basically the third case – new

alias function. The third option is that API is modified (just adding a parameter to one function might be sufficient). From module implementation point of view the best one, but from the rest of the rpmlib, this is the worst one, because every change in API means there is at least one more part of rpmlib to be changed as well.

The third approach is the most powerful, but also the most complex one. It consists in creating an API which would in fact simulate declarative language. User of this API tells it what information does he need, what are constrains of this information (e.g. only installed packages, packages with given name, etc.) and what should the order be (e.g. order the information by the time of installation). Based on these requirements, rpmdb module compiles a query for given database backend. Once compiled, it is executed and it is simple to iterate over results.

Chapter 7

Comparison

The comparison of old and new solution should start from the design of the database. The new database was designed to minimize issues of the old one. SQL part introduces a significant improvement over original solution. First of all it defines clear rules how to store the data. Whats more important, these rules are documented! Relational data model forces its basic concept, which has potential to improve the performance significantly – atomic information. Thanks to this, the information stored in rpmdb is much more effectively divided into small pieces. It is possible to work with these pieces independently on other information. That reduces the amount of data which has to be transferred between HDD and memory. Also if any information about package is needed which isn't in the database, there is secondary data source and that is directory structure containing every available information about installed packages. It is stored in very effective directory structure which reduces information lookup time. Directory structure in this concept is basically an index implemented directly by operation system. This concept can be also extended and another indexing parameters (not just ID and NEVRA) can be added. In case system administrator wants to work directly with RPM database for some reason (e.g. find and grep some information), it provides much more human-friendly face, especially when plain text files are used. Both solutions offer two-level design of the database. First level as a complete information about packages and the second level used as cache storing the most important information. This cache is far better in new solution because of its extent. In the old solution it is represented by secondary databases, which contain only a fragment of information that should be indexed today. The new solution on the other hand contains much more information, and yet it is better ordered thanks to relational data model. This only shows that this design greatly emphasize the significance of cache to make lookup faster. Also there is already mentioned possibility to look up records in primary database not only by key, but also by NEVRA. Together with easy creation of yet another indices this makes great shortcut to load the header if it is needed.

The new concept has also its disadvantages. In section 2.6 an analysis of disk space related issues was provided. The new solution will clearly take more space and the overhead will be also greater because of the expanded cache in a form of SQL database. Cache here has been measured by `du -sh <database>`, where `<database>` is a path to SQL database where all packages are installed. This is natural measurement of overhead, since data stored in header files. have no redundancy at all (except for internal filesystem structures). The overhead of new solution is *650 MiB*.

The last part of comparison in implementation of the program build on the top of the database. As it could be seen, rpm 4.7 has very complex code which is difficult to

comprehend, sometimes even for skilled developer. This has changed significantly since the development of this thesis has begun. But to reach a level when the code is easy to comprehend an easy comprehension of all concepts is required. The implemented code shows where the hardest part is – transformation of current data format into something more developer-friendly. If this new design is accepted and everything essential is implemented, this is where the development should continue – plain text format of both database and package headers. In both current and new solutions, the database format can be modified very easily. Although in the new solution, it offers much cleaner way in a form of SQL queries. Again it would lead to more straightforward code, which was the main goal of this thesis. The code of `query` program revealed the biggest contribution of this solution. By using SQL database as backend it is possible to achieve code several times simpler and cleaner than the current one. This claim is also confirmed by program implementing the deletion of the package. Here the effect would be much higher if the database engine was InnoDB and consistency checks were utilized. On the other hand, this would have to be very well commented in order to avoid obfuscation (code handling one situation split in two places – library and database schema definition). Another advantage over the original solution is platform independence. As it was described in section 3.1, some operations dealing with byte order and variable size exist in current implementation. These operations won't be necessary any more, because MySQL will be handling them. Another part of RPM might need these operations (e.g. loading of package header files), but that's not up to this thesis to analyze. From the API point of view, the new implementation offers cleaner API with open possibility of simple extension any time it is needed.

Chapter 8

Conclusion

Analysis provided in first chapters of this thesis is very useful not only for further design of new RPM database. Few issues were discovered in the code of RPM itself. Some of them were already reported and fixed, while others are still in the phase of debugging. There is also quite large amount of information about repository content. This information can be useful even for people who aren't working for RPM, but are responsible for planning the direction of Fedora development. Pieces of information about current content and future estimations are present through the entire thesis. Based on them, some suggestions are formed to improve packaging policy (of course supported by RPM itself) in the entire Fedora distribution.

Chapter 4 tried to bring some new ideas from competing package managers. Some improvements and ideas were indeed used further. Also the idea of using SQL database engine is very important because after all this is what brought most of the improvements. Comparison of MySQL embedded and SQLite was provided as a proof of MySQL convenience as RPM database engine instead of SQLite, which is greatly preferred by many people.

Luckily the mysql-embedded package doesn't have another mysql packages as a dependency, so RPM potential dependency isn't that much of an issue. Despite this it might seem to be too large dependency to be included in core system for server distributions. But considering that current RPM needs libdb-4.7 it can be considered as acceptable replacement.

The main contribution is that the new design is promising alternative to current RPM database. The implementation provided in chapter 6 is only proof-of-concept work and thus the main focus should be on the design, which is described in chapter 5. The design has many advantages over the current one and once a suitable API is completed on top of it, it has a good chance of being the next generation RPM database. In such case this thesis will be a foundation for future development and implementation.

Bibliography

[1] Appropriate uses for sqlite. http://www.sqlite.org/whentouse.html.

[2] Gentooportage – portage is a package management system used by gentoo linux. http://www.gentooportage.info/, May 2006.

[3] Pisi database schema. http://svn.pardus.org.tr/uludag/trunk/pisi/doc/pisi-db.xmi, September 2006.

[4] Rpm 4.4.2.2 api documentation. http://rpm.org/api/4.4.2.2/pages.html, December 2007.

[5] Archlinux propaganda: Arch linux is good, quite good. http://archpropaganda.blogspot.com/2008/07/arch-linux-is-good-quite-good.html, July 2008.

[6] Rpm container file format specification. http://rpm.org/wiki/DevelDocs/FileFormat, September 2008.

[7] Archlinux – arch build system. http://wiki.archlinux.org/index.php/Arch_Build_System, December 2009.

[8] Sqlite fts3 virtual table. http://dotnetperls.com/sqlite-fts3, January 2010.

[9] MySQL AB. Mysql embedded data sheet. http://www.mysql.com/oem/mysql_embedded_server_ds.pdf, November 2009.

[10] MySQL AB. Mysql reference manual. http://dev.mysql.com/doc/refman/5.0/en/bdb-storage-engine.html, October 2009.

[11] MySQL AB. Mysql reference manual. http://dev.mysql.com/doc/refman/5.4/en/libmysqld.html, November 2009.

[12] Edward C. BAILEY. *Maximum RPM.* Red Hat Software, Inc., Durham, NC, 2000.

[13] Jonathan CORBET. Ols: An introduction to conary. *LWN.net*, July 2004.

[14] Florian FESTI. Rpmdb performance (rpm developer discussion). http://lists.rpm.org/pipermail/rpm-maint/2009-August/002460.html, August 2009.

[15] Eric FOSTER-JOHNSON. Rpm guide. http://docs.fedoraproject.org/drafts/rpm-guide-en/, 2005.

[16] Jan KOMÁREK. Teorie relačních databází: Normalizace. http://www.manualy.net/article.php?articleID=13, August 2007.

[17] Oracle. Getting started with berkeley db. http://www.oracle.com/technology/documentation/berkeley-db/db/gsg/C/index.html, December 2009.

[18] rSync. Conary api documentation. http://cvs.rpath.com/conary-docs/, October 2009.

[19] Luke MACKEN Seth VIDAL and James ANTILL. Source code of createrepo utility. http://createrepo.baseurl.org/git/createrepo.git, December 2009.

[20] Eric TROAN and Marc EWING. Source code of rpm 4.7.0. http://rpm.org/releases/rpm-4.7.x/rpm-4.7.1.tar.bz2, July 2009.

[21] Jaroslav ZENDULKA and Ivana RUDOLFOVÁ. Databázové systémy ids - studijní opora, July 2006. Internal document of Faculty of Information Technology, Brno University of Technology.

List of attachments

A. DVD

B. Profiling graphs