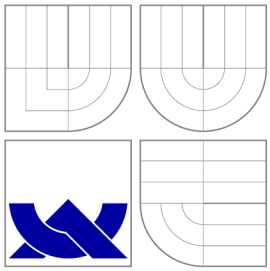
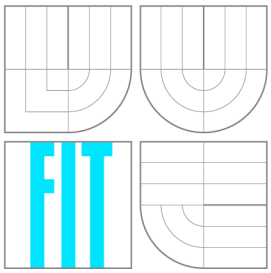


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁZEV PRÁCE

UNIFIED SOFTWARE DATABASE FOR RPM BASED SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

JAN ŠILHAN

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN ZELENÝ,

BRNO 2014

Abstrakt

V GNU/Linuxovém prostředí je nepřeberné množství možností, jak instalovat aplikace. V dnešní době existuje spousty nástrojů, které mají na starost různé části systému. Linuxové distribuce mají hlavního správce balíčků a populární programovací jazyky mají také vlastního správce balíčků. Všechny tyto nástroje si udržují informace o nainstalovaném softwaru, a proto si každý z nich spravuje vlastní databázi s redundantními metadaty o balíčcích. Záměrem této práce je analyzovat úložné prostory, identifikovat případy užití správců balíčků v distribuci Fedora a následně navrhnout a implementovat jednotnou centrální softwarovou databázi na systému.

Abstract

In GNU/Linux environment there are many ways of installing software. At this moment we have multiple tools for managing some specific parts of the system that overlap each other. Linux distributions have main package management system and the popular programming languages have also its own package manager. These all have to keep track of installed software. Thus every package manager maintains its own private database with redundant package metadata. The motivation of this thesis is to analyze storages and identify common use cases of package managers on Fedora distribution; design and implement one central Unified Software Database on the system where all information about packages could be stored.

Klíčová slova

RPM, Swdb, RPMDB, USD, spávce balíčků, databáze, balíčky, DNF, Yum, PackageKit

Keywords

RPM, Swdb, RPMDB, USD, package management, database, packages, DNF, Yum, PackageKit

Citace

Jan Šilhan: Unified Software Database for RPM Based Systems, diplomová práce, Brno, FIT VUT v Brně, 2014

Unified Software Database for RPM Based Systems

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Zeleného

.....
Jan Šilhan
May 28, 2014

Poděkování

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

Contents

Chapter 1

Introduction

In *GNU/Linux* (later only as Linux) environment there are many ways of installing software. The most common methods are compiling from source code or installing a package. Package contains metadata about itself, which is useful for a package manager. What a package manager does is it downloads, sets up and installs the software and all its needed dependencies. So instead of downloading a dozen source code archives manually and compiling all of them you can install everything executing one simple command.

At this moment we have multiple tools for managing some specific parts of the system that overlap each other. Linux distributions have main package management system (RPM, Deb, ...) and the popular programming languages have also its own package manager. These all have to keep track of installed software. Thus every package manager maintains its own private database with redundant package metadata that takes additional space on disk.

The motivation of this thesis is to design and implement one major *Unified Software Database* (further referred to as USD) on the system. The aim is not to unify the package managers but offer them one central place where all information about packages could be stored. Each of them will have an opportunity to write its own package related data there. The solution I am trying to find is primary for *Fedora* and *RHEL* but simultaneously easily portable for other Linux distributions.

In this document is made up USD API that covers the most common scenarios that happen during the package installation, upgrade or removal process; based on deep analysis of Fedora package managers' use cases. The implementation of USD will be used in production, starting with DNF 2.2.3.

In chapter 2 many possibilities of installing applications on RPM system will be covered. I will go from plain RPM package installation through advanced command line tool for solving/downloading package dependencies to graphical user interface applications.

The use cases and currently utilized storages in RPM-based package managers that should USD take into consideration will described in chapter 3. *RPMDB*, main RPM database, is introduced in section 3.1.1. Its analysis will be essential for better understanding RPM package format. The breakdown of Yum storages in section 3.3.2 will influence the design of USD most.

The next chapter 4 is all about the design of USD. The requirements for USD from package managers will be listed in section 4.1. Design scheme resulted from previous analysis will be introduced in section 4.2. The next section (4.3) explains advantages of USD that comes from redesigning history database and yumdb with extra features added. The API of USD is discussed in section 4.4. Later on wide variety of USD database engine

candidates will be compared and one of them selected (section 4.5). Last item (section 4.6) in this chapter will go through implementation details that occurred during USD build.

Following chapter cover yumdb and USD benchmarks of read, write and erase operations and total disk space consumption. The summary of these test cases can be seen in section 5.6.

The last chapter 6 summarize the result of this document, offer ideas for improvement and outline future of USD.

Chapter 2

Package management

Package management is a paradigm of managing software in discrete parts, called packages. It is an unnecessary piece of Linux operating systems and the region where Linux distributions differs the most.

Packages can be composed of precompiled binary or source code files. Apart from all the various files, documentation, and configuration information, the more sophisticated packages can contain metadata, such as the software's name, description of its purpose, version number, vendor, checksum and a list of dependencies necessary for the software to run properly in structured format.

Dependencies are one of the most important parts of the package management system. A dependency occurs when one package depends on another. The package manager ensures that dependencies are honored when upgrading, installing or removing packages. The biggest advantage is that each package knows, by defined dependencies, which package has to be installed before. That way one can install or upgrade the desired application without manually downloading the required packages. The process during install, upgrade or removal that mark what package have to be installed or upgraded, is called dependency solving (*depsolving* later).

The package management system typically maintains a database of software dependencies and version information to prevent software mismatches and missing prerequisites. Upon successful installation metadata of package is stored. Each package management system is often linked to one concrete package format they can operate with, thus they have different local database scheme.

Package management systems are popular between specific programming languages. Packages are often modules, written in a particular programming language, that are all saved in the respective directory. Some packages can also be installed globally as a regular application. Package managers for programming languages are targeted for developers to set up their projects with all dependencies. Representatives include *ruby gems*, *easy_install*, *maven* or *npm*. This category of package managers will not be discussed in this thesis as they cannot replace RPM tool when most software in Linux is written in compiled languages like C or C++ and they are bound to certain language they can handle. Moreover, these tools don't have advanced features of Yum/DNF like history commands and don't have *groups* (packages that could be installed/removed as a whole by group name reference). As a result, Yum/DNF have a much more complex database than pip, maven, npm and ruby gems all together.

2.1 RPM Packages

An RPM Package is described by the *spec file*, where all the package information is kept. The package is identified by *NEVRA* (name, epoch, version, release, architecture). The *Requirements*, *Provides*, *Conflicts* or *Obsoletes* dependencies can also be declared in the spec file. The reldep is a tuple consisting of the type of dependency, package name and optional signs (=, <=, >=, <, >). The sign can specify the version of dependent package. Every package can also provide another feature (virtual package), hence the RPM system is file-based [1].

Triggers provide a way for one package to take action when the installed status of another package changes. A trigger is a script you define in your package's spec file that gets run by the RPM system when the status of another named package changes [1].

2.2 Package managers for RPM packages

The following subsections discuss package managers dealing with RPM packages. At the lowest level in the package manager's hierarchy (picture 2.1) is the RPM tool that is necessary on Red Hat/Fedora platforms. Package managers on top of RPM are adding the functionality of downloading dependencies or treating a set of packages as a single application.

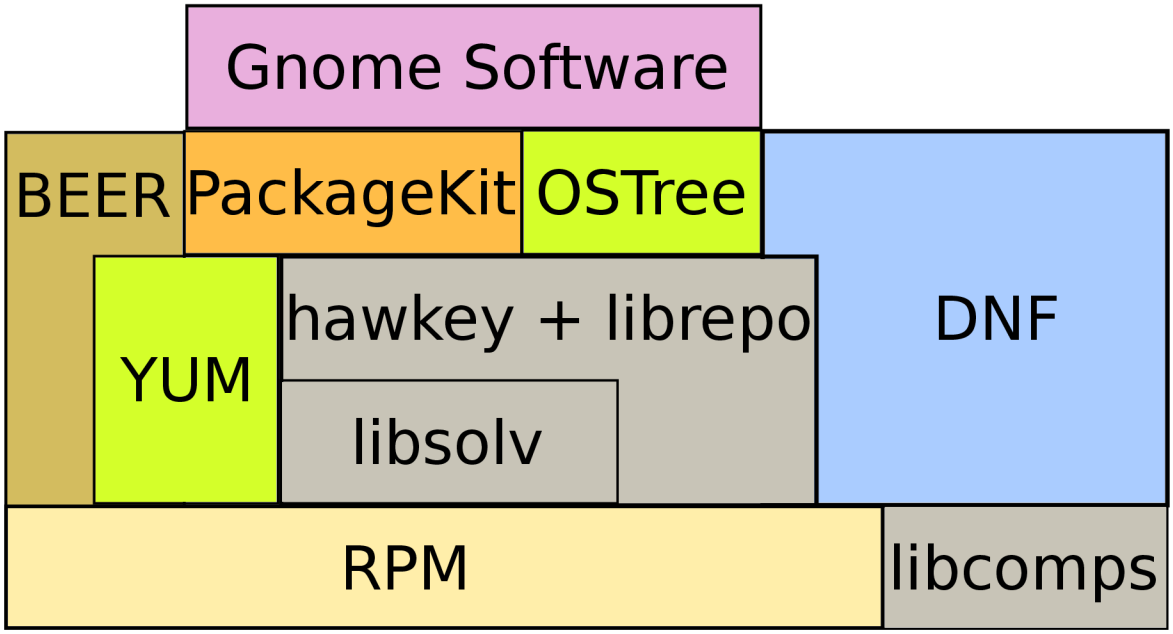


Figure 2.1: Package managers hierarchy in RPM-based systems

2.2.1 RPM

The *Red Hat Package Manager* (RPM) is an open packaging system, which runs on Red Hat Enterprise Linux as well as other Linux and UNIX systems. It is responsible for verification that installed applications have been installed correctly. The RPM software maintains a database on the system of all packages that have been installed, and documenting which files those packages have installed on the system.

For installation, reinstallation or update, all desired packages have to be already on the file system or the URL of the package has to be provided. RPM Packages can be found and eventually downloaded for instance from *rpm.pbone.net* or *rpmfind.net*.

2.2.2 Yum

Yum is a well-known command line automatic updater and package installer/remover for RPM systems. It automatically computes and downloads dependencies. All the hard work is done in Yum itself, utilizing RPM for pure package set installation. It makes it easier to maintain groups of packages without having to manually update each one using RPM. Yum has a plugin interface for adding simple features. It can also be utilized from other python programs via its module inteface. In Fedora 20 it is considered as deprecated in favor of DNF.

2.2.3 DNF

DNF (Dandified Yum) is the successor of Yum with the same code base (which is being reduced) but utilizes the Libsolv library (see 3.5) as the dependency solver. The project was set up because of hard maintainability of Yum. DNF professes the divide-and-conquer principle, therefore it relies on many small libraries rather than doing all the stuff itself. The main goals of the project are:

- using a SAT solver for dependency resolving
- allowing us to eventually use the same solver in RPM too
- strict API definition for plugins
- strict API definition for extending projects (Anaconda)
- leaner codebase than Yum, allowing for easier maintenance
- better performance and memory footprint.

2.2.4 BEER

BEER stands for Better Extension for RPM and is an alternative implementation of updating/buildroot creation and dependency resolving tool written in C. The main aim is to provide an alternative to Yum, which is very slow and resource wasteful as a native Python application. BEER utilizes per-repository package caching, so reinstallation or updating of chroot is faster. RPM packages are held in local filesystem cache hence they don't need to be downloaded from web repository repeatedly [13]. The official project repository has not been touched for three years, though.

2.2.5 Yumex

Yumex is a graphical user interface for the Yum package manager. It is a graphical front-end similar to Synaptic (uses Debian's apt tools). Yumex allows you to install new applications as well as update or delete existing ones via GUI. Aside from packages, user can manage repositories and groups. Fedora is the distribution of choice for Yumex, but it can be run on any distribution that uses Yum. At the time of writing this, Yumex is preparing for switching from Yum to DNF backend.

Yumex was created with these goals in mind:

- To create a advanced Yum GUI with a lot of features for both the novice and the advanced users
- To make it easy to update, install and remove application.
- To make it easy to find applications.
- Give the user a choice to see what is going on between behind the curtains.
- To show the power of Yum
- Give the user access to some of the more advanced features of Yum in an easy way.

2.2.6 PackageKit

PackageKit is designed to provide a consistent and high-level front-end for a number of different package management systems [5]. It is a lightweight daemon, that is creating unified layer between Graphical front-ends and software management tools like Yum, Zypp, uRPMi, etc. PackageKit's main features are:

- Boot time security updates
- Allowing unprivileged users to install software in a corporate build
- Opening unknown file formats
- Removing dependencies for files

2.2.7 Gnome Software

In newer releases of Gnome, the Gnome Software application for installing and removing packages from graphical user interface is present. The concept is similar to Ubuntu Software Center. In contrast to Yumex, Gnome Software is application-oriented and hides the details about packages. It presents an application to the user by providing screenshots rather than package dependencies. Software uses a plugin architecture to separate the frontend from the technologies that are utilized underneath [4]. Gnome Software's capabilities are:

- View installed applications
- Remove installed applications
- View available application updates
- Install available updates

- Find new applications
- Install new applications
- Find an application to handle a specific type of file
- Installed apps and updates should be available when offline

2.2.8 OSTree

OSTree is a tool for managing bootable, immutable, versioned filesystem trees. It is not a package system; nor is it a tool for managing full disk images. Instead, it sits between these two levels, offering a hybrid tree/package model, thus benefiting from the advantages of both [3].

A fundamental goal of the project is to enhance the ability of existing package systems like Debian packages or RPM (RPM-OSTree). The mechanism how it works is replicating base tree via OSTRee and then adding packages on top of it. Unlike other similar projects (Google ChromeOS autoupdate, Ubuntu Image Based Updates) it is more space efficient by taking just snapshots (diff of binary files) therefore it doesn't duplicate entire filesystem of operating system [15].

2.2.9 Software Collections

Software Collections (Scl) was implemented to give users the opportunity to run multiple versions of the same program. In this case, two directories are not merged into one, but every application with all its dependencies (together collection of packages) are saved into one directory located in */opt/scl/<collection of packages>/root*. Scl works with RPM packages. It changes the software installation directory by extending the package spec files by additional macros. A program within given collection can be run via `scl --enable command <collection of packages>`. That will change all system variables to emulate root path in */opt/scl/<collection of packages>/root* direction [10].

Chapter 3

Database In package management Applications

In the previous chapter we got acquainted with the overall technologies of package management systems. Now we will focus on how data are stored in Fedora package managers and common database operations. Picture 3.1 shows the interaction of package managers and access to database storages. Note that the picture is cleared of the details how metadata of available packages are obtained by Yum and DNF.

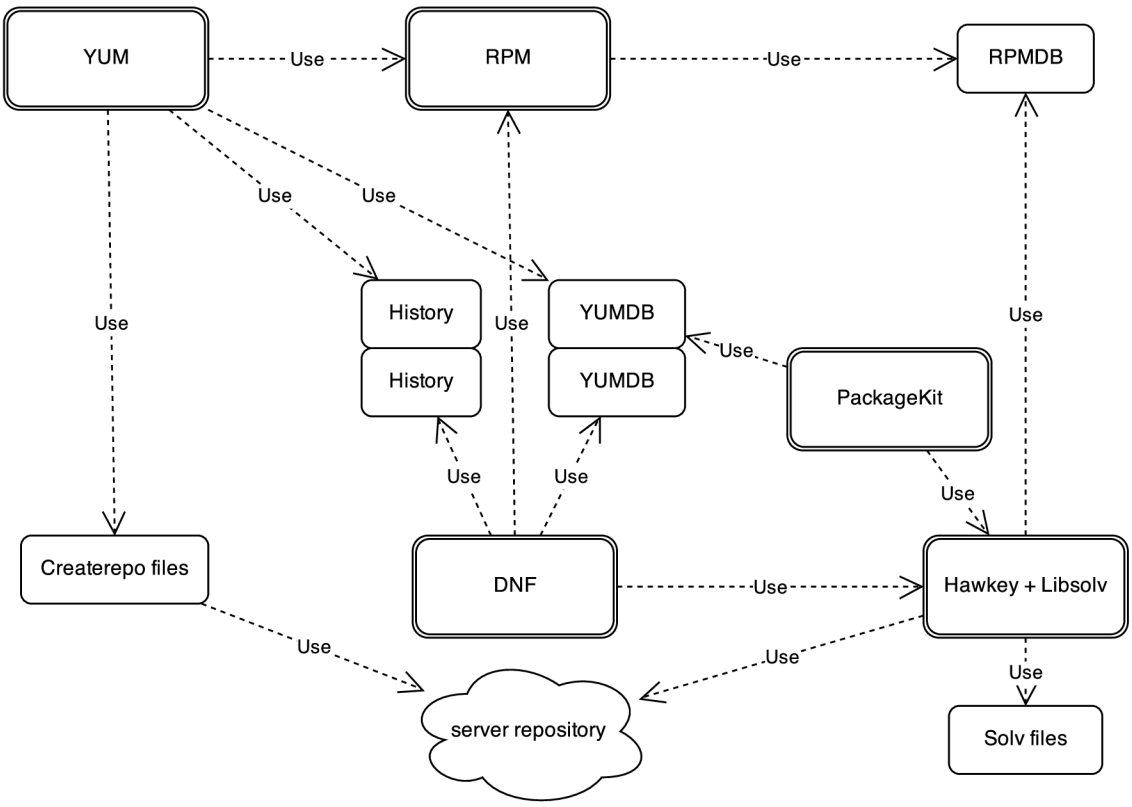


Figure 3.1: Package managers interaction between storages nowadays

3.1 RPM Use Cases

As it has been said, RPM check whether dependency of installing package are fulfilled. It does not read all installed packages, but just the ones it needs for dependency checking and doesn't download any new package by its own.

Another RPM function is detecting file conflicts, that is what the *Basenames* index is for. It does not contain the full path because it has to consider directory aliasing. The file conflict detection needs the checksums of all files. All the checksums need to be stored then. This action takes the most space of all package related attributes in the database.

RPM is designed to be queried easily, making it possible to search this database to determine what applications have been installed on the system and to see which packages have supplied each file on the system.

Here is a comprehensive list of most common RPM actions [19]:

- 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 look-up 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

3.1.1 RPMDB

RPMDB is the main storage maintaining package metadata and is designed in the first place for RPM needs. That is keeping all the information about installed packages (the header) and searching fast for a record of certain package by file, group or reldep (Berkeley DB database part). The header structure contains three parts: header record, one or more

header index record structures and data for the index record structures. The header record identifies this as the RPM header. It also contains a count of the number of index records and the size of the index record data. Each index record uses a structure that contains a tag number for the data it has [9].

Whenever you install or remove packages, the RPM system assigns *transaction ID* to the set of packages and *installed ID* to a single package. The transaction ID is a Unix time stamp. All the packages installed at the same time are given the same transaction ID. This means that you can perform operations on a set of packages, the packages that were installed together.

In the contrast, installed ID is unique identification of installed package. For reading from stored header, installed ID is being used.

RPM is supposed to hold the database write lock for short periods of time, when a new package header is written into the db, or a header is deleted. There is also the transaction lock that is held for the complete transaction. Berkeley DB does all of the locking internally.

Berkeley DB is key-value storage utilized for a couple of index databases (*Providename*, *Conflictname*, *Obsoletename* and *Requirename*). Those map the *name* part of a dependency to a (*installed ID*, index number of the dependency) tuple. RPM splits name of files, contained in package, into (*dirname*, *basename*) tuples. In the header it is stored as follows: *dirnames* (array of strings), *dirindexes* (array of integers) and *basenames* (array of strings).

Here is the brief overview of Berkeley DB files listed in */var/lib/rpm/* directory [19]:

- Basenames (B-Tree) – File name (not a path, only a name) is a key and values contain pairs of (*installed ID*, basename index). This index is very closely related to *Dirnames*
- Conflictname (B-Tree) – Name of conflicting package is a key here, pairs of (*installed ID*, conflictindex) create values
- Dirnames (B-Tree) – Index containing paths to files listed in *Basenames*. Path is a key and values are (*installed ID*, directory index)
- Group (B-Tree) – Valid name of a group is a key and values are created by couples: (*installed ID*, 0)
- Packages (Hash) – Package name is here as a key and values are: (*installed ID*, 0)
- Providename (B-Tree) – Same as *Conflictname* for provide tag.
- Provideversion (BTree) – Contains versions of packages listed in corresponding records of *Providename* as a value
- Requirename (B-Tree) – Index is corresponding with *Conflictname* and *Providename*
- Requireversion (B-Tree) – Index corresponds with *Provideversion*
- Triggername (B-Tree) – Key is unique name of a package trigger. Values contain couples of (*installed ID*, triggerindex)

The main problem with RPMDB is the inability to write application's custom data there. Thus every package management system has its own separate storage with many redundant data with regard to the master database.

3.2 BEER Use Cases

BEER uses Yum repository configuration files to download RPM packages from various repositories and resolve dependencies between the user-specified packages to be installed. Since BEER is a low-level tool, the location of RPMs as well as the desired chroot can be specified directly on a command line without any need for Yum to be installed.

BEER utilizes cURL for package downloading and RPM for transaction checks and the chroot for installation itself. It can be used on any RPM-based Linux OS, mainly on Fedora Core/Red Hat ones.

This tool is mentioned only for complete enumeration of applications founded in RPM-based systems. BEER is not as feature-rich as Yum and so doesn't use any additional storage. Moreover it is just a dependency solver that is no longer maintained and useless in the new age of Libsolv (see 3.5). For that reason it will not be entertained in USD design.

3.3 Yum Use Cases

Both Yum and DNF are using YumDb and history database, but not at the same path. Hence they are not synchronized and don't know about each other's transactions. When some packages were installed outside of DNF/Yum for example by PackageKit, they lose track of these packages. That could lead to showing warnings that the database was altered by another application or to the inability to run history commands on packages installed in another way. Note that metadata about all the installed software is always kept in RPMDB either way.

3.3.1 SQLite Files Of Createrepo

Createrepo, a tool that can create repository in directory, can output also compressed SQLite database files, that can be fetched by Yum. It stores information used for Yum dependency solving that is saved during update process in following directory */var/cache/yum/<arch>/<fedora_version>/<repo_name>/gen/*. The most important files containing package metadata tables are listed below.

- `primary_db.sqlite` – conflicts, obsoletes, provides, requires, db_info, packages
- `filelists_db.sqlite` – db_info, packages, filelist
- `pkgtags.sqlite` – packagetags
- `other_db.sqlite` – changelog, db_info, packages

There is list of tables from SQLite files. The columns in bold are indexed.

- Conficts, obsoletes and provides tables describes relation between packages. The fields for all of them are the same.
  - **name** – name of the package
  - flags – disjunction of *reldep* signs (see 2.1)
  - epoch – The Epoch tag in RPM is to be utilized only as a last resort. It is sometimes necessary to use an Epoch to handle upstream versioning changes or to ease transition from third party repositories [14].

  - version – version of library/application that package represent
  - release – release of the package, e.g. operating system where was the package built, plus hash or id of the build.
  - **pkgKey** – an unique package id in the table

- In requires table all equally named fields from conflict, obsoletes and provides tables have meaning just as the previous ones:

  - **name**
  - flags
  - epoch
  - version
  - release
  - **pkgKey**
  - pre – signs whether package is required before installation or not (*BuildRequires* or *Requires* tag in spec file)

- db_info table:

  - dbversion – lowest version of Yum's metadata parser required
  - checksum – 256 bit hash of all packages

- packages:

  - **pkgKey**
  - pkgId – package hash – an unique package identifier across *filelist_db* and *primary_db* files
  - **name**
  - epoch
  - version
  - release
  - summary – short summary what is the package used for
  - description – more detailed information of the package's purpose
  - url – usually project's homepage, not direct url for package download
  - time_file – package's last modified UTC time
  - time_build – UTC time when package was built
  - rpm_license – license of the package
  - rpm_vendor – who is responsible for distributing this package (for fedora main and updates repository that is *Fedora Project*)
  - rpm_group – which category the package belongs to (e.g. texlive package is in *Applications/Publishing*)
  - rpm_buildhost – commonly the server where was the package built for target architecture using *fedpkg* tool

  - rpm_sourcerpm – name of *.src.rpm package from which was architecture dependent package built
  - rpm_header_start – start of rpm header in binary blob used for identification and verification of package
  - rpm_header_end
  - rpm_packager – who built the package
  - size_package – size of RPM package with spec file and patches
  - size_installed – size of all extracted files
  - size_archive – size of source archive
  - location_href – last part of url where the package can be downloaded with concatenation of *location_base*
  - location_base – Url prefix to package download path. If it is empty, repository url is used
  - checksum_type – supported hash algorithms are MD5, SHA1, SHA256, SHA384 and SHA512
- files:
  - **name** – absolute file/directory path
  - type – could be either "dir" or "file"
  - **pkgKey**
- filelist:
  - pkgKey
  - **dirname** – name of directory containing package files
  - filenames – list of files from *dirname* separated by '/'
  - filetypes – It is a string of size of total filenames count. It is gaining value 'f' for file or 'd' for directory at the same index position as the filenames string. E.g. for "script.sh/dir" as filenames is filetypes equal to "fd".
- changelog table holds release news.
  - pkgKey
  - author – author of build, enchantment or bugfix commit
  - date – date of commit
  - **changelog** – commit message
- packagetags:
  - name – package name
  - tag – category membership of package (for example: Utility, Application, GTK, Security, ...)
  - score – calculate how much application is associated with given tag (ranges from 1 to 10)

3.3.2 YumDb

Yumdb, also called the database database, is utilized for storing additional package metadata that couldn't be stored inside RPMDB. It proposes free space for data of Yum and all Yum's plugins. None of the information stored there is critical to performing its function but it enhances the user experience and makes it possible to know more about the context in which a package was installed. For storing additional package related information no database engine is used. Yumdb take advantage of file system with tree structure based on unique path for each package according to it is NEVRA – */var/lib/yum/yumdb/p/<checksum>-<packagename>-<version>-<release>.<architecture>/<keyname>* [8].

Actually it uses the same key-value principle as Berkeley DB, whereas in this case file system is doing job of database, the key is represented by path to file and value is stored as file content.

The disadvantage of the file system as database is that searching is rather slow for complex queries. Fortunately Yum doesn't use them. This solution has more advantages:

- Each package data are in isolation. If the key for some package is broken, nothing else should be affected.
- Simple realization with no additional database dependency
- Have interoperability by ability to perform get or set operations from any language without having to access the Yum API (that is what PackageKit silently does).

Here is the list of all required keys (files) stored in package name space:

- from_repo – the name of the repository from which the package was installed
- from_repo_revision – repository, revision or ctime for a local package
- from_repo_timestamp – repository, timestamp or mtime for a local package
- reason – reason for installing this package (user or dep)
- releasever – releasever of the system at the time the package was installed
- installed_by – the loginuid of the user who first installed this package
- changed_by – the loginuid of the user who last installed this package

The optional keys are:

- checksum_data – the value of the checksum for the installed package.
- checksum_type – the type of the checksum for the installed package (md5, sha1 or sha256)
- command_line – the command line used to install this package
- group_member – set by Yum if a package was installed as part of a *group install*
- installonly – not set by Yum, but looked at to see if installonly packages should be automatically removed
- origin_url – the url that the package was downloaded from

File system storage seems to me like a waste of disk space since every key needs a separate file. One ext4 block takes a minimum of 4kB regardless of the fact that the file content is only a few characters long.

3.3.3 History database

The reason why Yumdb and the history database are separated is that they hold different data. Yumdb represents the state of currently installed packages on the system while the history database maintains a log about transaction sets of installed, removed or updated packages in a bulk. Even unfinished transactions with all occurred errors are saved. The history database is crucial for DNF/Yum history commands (redo states of packages). The database also incorporates configuration data related to all transactions made. These are not exactly in the database; rather, they are saved as key-value pairs in plain text in the */var/lib/DNF/history/<date>/<transaction_id>* directory where transaction_id is linked to a transaction in sqlite. Their relationship is of the cardinality 1:1. There are two configuration files: config_main (global configuration of package manager) and config_repos (concatenated configurations of repositories into one string).

The whole history database in DNF is located in */var/lib/dnf/history/history<date>.sqlite*. Yum/DNF save snapshots of them time by time. They consist of the following tables and columns:

- pkg_yumdb – Assigning data from Yumdb to package. Table is made universal to maintain any key value.
  - pkgtupid – identifier of package translated from NEVRA
  - yumdb_key – is command_line, from_repo, from_repo_revision, from_repo_timestamp, installed_by, changed_by, reason or releasever
  - yumdb_val
- pkg_RPMDB – Assign data fetched from RPMDB to package. Same concept as in previous table for any key.
  - pkgtupid
  - RPMDB_key – could be one of license, reason, url, packager, size, buildtime, buildhost, sourcerpm, vendor, committer or committime
  - RPMDB_val
- pkgtups – stores basic identification information about package that other tables references
  - pkgtupid – identifier of package
  - name – name of the package
  - arch – package architecture
  - epoch – package epoch
  - version – package version
  - release – release of the package
  - checksum – package checksum
- trans_beg – Transactions that are processed or unfinished transactions. Row is filled by Yum/DNF before running RPM's transaction for package set.
  - timestamp – start of transaction in UTC

  - RPMDB_version
  - loginuid – user who initiated transaction
- trans_end – stands for completed transactions
  - tid – transaction identifier
  - timestamp – end of transaction in UTC
  - RPMDB_version
  - return_code – result of transaction. If 0 then process was successful otherwise failed
- trans_cmdline – stores command line history
  - tid
  - cmdline – exact DNF/Yum command line that triggered transaction
- trans_data_pkgs – connecting packages to transaction
  - tid
  - pkgtupid
  - done – column can be TRUE or FALSE whether package *state* was accomplished or not
  - state – could be one of (Install, Reinstall, Update, Downgrade, Erase, Reinstalled, Updated, Downgraded, Obsoleted)
- trans_error – logged errors that occurred during transaction
  - tid
  - msg – error message from RPM
- trans_prob_pkgs – packages that run into problems during during Yum/DNF dependency solving
  - rpid – ID of problem package
  - pkgtupid
  - main – could be either TRUE or FALSE based on whether this package caused that error.
- trans_rpmdb_problems – transaction errors from RPM
  - rpid – ID of problem package
  - tid – transaction ID
  - problem – problem summary message
  - msg – more detailed information of error
- trans_script_stdout – output from stdout during execution of transaction
  - tid – transaction ID
  - line – message

- trans_skip_pkgs – logged packages that was skipped during transaction
  - tid – transaction ID
  - pkgtupid – package ID
- trans_with_pkgs – programs that were involved in installation/removal/upgrade transaction process. That is always 'DNF/Yum' and 'RPM'.
  - tid – transaction ID
  - pkgtupid – package ID

Data in the current history database are not saved permanently. They live until a new package of the same version is installed. After that the attributes and installation data of the new package replace the current records. Tables pkg_RPMDB and pkg_yumdb from the history transactions database hold the data for an additional period between package removal and its re-installation. This is in contrast to rpmdb, where data are deleted after package removal. Yumdb data last as long as metadata copied to pkg_yumdb. The only difference is that yumdb data can be changed anytime. Pkgtups items are unique and permanent because thay are used as references to not-always-present package. The same will remain in the new concept. The diagram 3.2 shows for how long data persists in each database for a package installation, removal and reinstallation that happened over a longer period. These operations weren't executed in one transaction but in three separated transactions (installation, removal, installation) for a package with the same NEVRA. The complete database structure can be seen in picture 3.3.

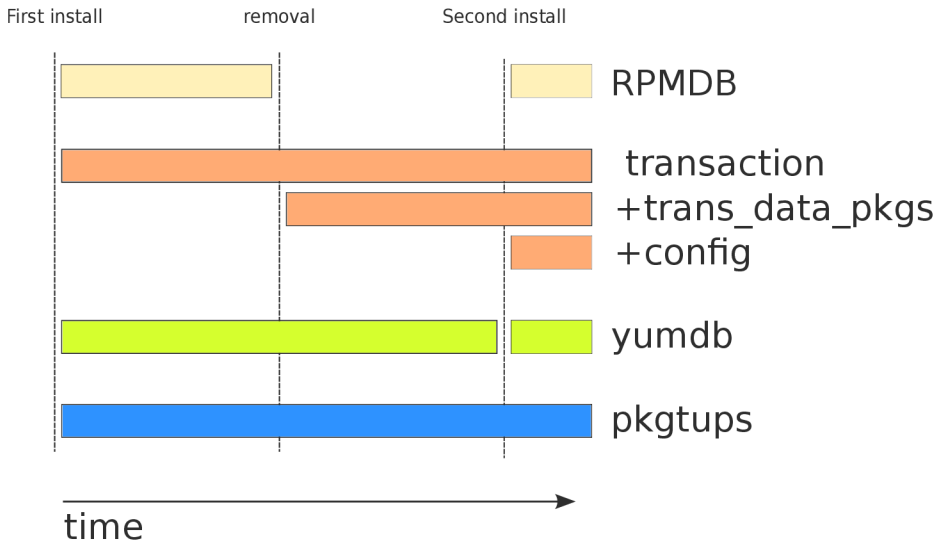


Figure 3.2: Data persistence in storages

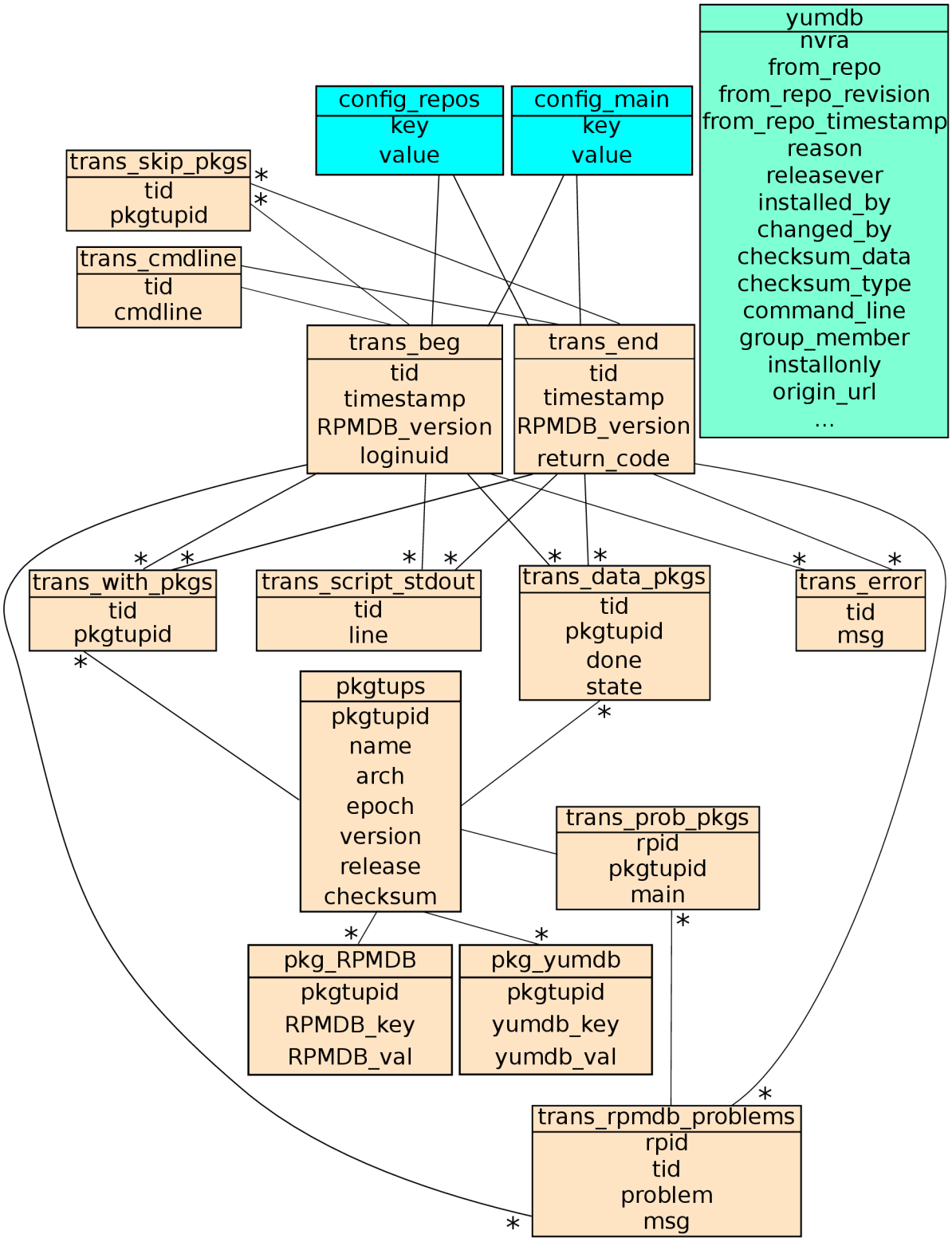


Figure 3.3: database scheme in Yum and DNF nowadays

Historydb API

This section enumerates historydb methods to get idea how unnecessary complex it is. There are not fully described all parameters the function takes.

The historydb python API utilized by Yum and DNF is illuminated below:

- `class YumHistoryPackage` – class holding the reference to package id
  - `def __init__(self, name, arch, epoch, version, release, checksum=None,history=None)` – constructor
  - `def __le__(self, other)` – Compares package attributes in following order: by architecture, whether is installed or not and eventually by the time of installation.
  - `def __getattr__(self, attr)` – Load rpmdb attributes from the historydb.
  - `def ui_envra(self)` – Returns string representation of the package.
  - `def envra(self)`
  - `def nevra(self)`
  - `def nvra(self)`
  - `def returnIdSum(self)` – Returns checksum of the package from historydb.
  - `def verCMP(self, other)` – Compares two packages by name and version.
- `class YumHistoryPackageState(YumHistoryPackage)`
  - `def __init__(self, name,arch, epoch,version,release, state, checksum=None, history=None)` – Prepares package for transaction, i.e. sets state and mark it as undone.
- `class YumHistoryRpmdbProblem(object)` – class representing a rpmdb problem that existed at the time of the transaction
  - `def __init__(self, history, rpid, problem, text)` – constructs package problem
  - `def __cmp__(self, other)` – Compares YumHistoryRpmdbProblem objects by message first, then by package id.
  - `def packages()` – Get all YumHistoryPackage objects that caused the same error within transaction.
- `class YumHistoryTransaction` – handler for a history transaction
  - `def __init__(self, history, row)` – constructor
  - `def __cmp__(self, other)` – Compares two YumHistoryTransaction objects by time of the beginning and end of transaction; and by transaction id.
  - `def getTransWith(self)` – *get** methods are for accessing tables in relation with transaction table
  - `def getTransData(self)`
  - `def getTransSkip(self)`
  - `def getProblems(self)`
  - `def getCmdline(self)`

  - `def getErrors(self)`
  - `def getOutput(self)`

- `class YumMergedHistoryTransaction(YumHistoryTransaction)` – class used for merging transactions together
  - `def merge(self, obj)` – Tries to merge two transaction into one. For example from pkgA-1 => pkgA-2, pkgA-2 => pkgA-3, pkgB-1 => pkgB-2 and pkgB-2 => pkgB-1 package transformations becomes pkgA-1 => pkgA-3 and pkgB-1 => pkgB-1 (reinstall) transformations. When any transaction in path is missing, no merge is applied.

- `class YumHistory` – main handler for historydb
  - `def __init__(self, db_path, yumdb, root='/', releasever=None)` – constructor
  - `def close(self)` – Properly ends session.
  - `def pkg2pid(self, po, create=True)` – If create is True, inserts new record into database, otherwise gets package id and checksum from existing row.
  - `def trans_with_pid(self, pid)` – Stick in NEVRA representation of package managers that initiated transaction.
  - `def trans_skip_pid(self, pid)` – Inserts log of skipped packages in transaction.
  - `def trans_data_pid_beg(self, pid, state)` – Inserts record of package change.
  - `def trans_data_pid_end(self, pid, state)` – Mark package transformation completed.
  - `def beg(self, rpmdb_version, using_pkgs, tsis, skip_packages=[], rpmdb_problems=[], cmdline=None)` – Logs start of the transaction.
  - `def log_scriptlet_output(self, msg)` – Inserts RPM output from stdout during transaction.
  - `def end(self, rpmdb_version, return_code, errors=None)` – Signs transaction as done.
  - `def write_addon_data(self, dataname, data)` – Writes configuration data to the file.
  - `def return_addon_data(self, tid, item=None)` – Returns all configurations from the file.
  - `def old(self, tids=[], limit=None, complete_transactions_only=False)` – Return a list of the last transactions. Note that this includes partial transactions (ones without an end transaction) by default.
  - `def last(self, complete_transactions_only=True)` – Returns the last full transaction. Any incomplete transactions do not count if not specified.
  - `def sync_alldb(self, ipkg)` – Synchronize all the data from RPMDB and yumdb for this installed pkg.
  - `def search(self, patterns, ignore_case=True)` – Search for history transactions which contain specified packages. Returns transaction ids.

The paths of Yum and DNF are diverging, therefore DNF doesn't take advantage of all the historydb functionality (namely addons, RPMDB values, ...) because users no longer need it. In USD API some methods should be more generic and do not add more boilerplate into the code.

3.4 DNF Use cases

Aside from history database and yumdb, DNF stores additional information in *JSON* format. The items saved are the repositories that were explicitly marked by user as expired and groups of packages installed by DNF.

Sqlite files of available packages are not read by DNF directly. DNF queries for packages through Hawkey (the wrapper for Libsolv). DNF gets the result without any parsing and computation.

3.5 Libsolv

Libsolv is a fast dependency solver using SAT technology. At present it is included in Zypper and DNF. This library can handle many package formats including RPM and Deb [16].

One great benefit this library offers, is metadata cache files in Libsolv's own binary format (files with .solv extension, typically). That is only a small subset of package data synchronized with master database (RPMDB). Once in a while Libsolv gets all the properties from a RPMDB by reading headers as a blob. It does not fetch them through RPM API because of its low performance. Libsolv could create solv files from xml files downloaded by Librepo as well.

3.5.1 Libsolv Use Cases

Typical Libsolv operation figures out what provides required packages. *Provides* of all packages, and the other dependency data of many packages needs to be known. It doesn't make sense to do specific look-ups for single packages. Instead, Libsolv reads the relevant data of all packages and keeps all the relevant stuff in memory. It creates index structures in memory for fast dependency look-up by hash.

3.5.2 Solv Files Format

The *solv file* format is not a database, but a specialized serialization format for dependency solving. They are light and highly optimized for loading speed. That is why Libsolv is so fast. At a cost of a few hundred milliseconds, using the solv files reduces repository load times from seconds to tens of milliseconds.

The purpose of showing solv file is to outline how effectively data can be stored and that could potentially influence approach of storing data in Unified Software Database. Most of the file contains dependencies and they need to be unified. To each dependency is assigned a unique number, same dependency strings get the same number.

Solv file consist of header and five sections. Header contains SOLV signature (4 bytes), time stamp (when was last synchronization with master database - uint32), following quantity of all kinds of data, represented by *uint32* type each (ids, package relations, ids of directories, solvables, keys, schemata, flags, size of ids). The first part contains all relevant

strings from in package headers (package names, evr, target architecture, license, application group, etc.). Solv files are not compressed by any tool, however every single string can be found there no more than once. References to strings offsets are put into hash table for faster look-up. The next section contains tuple of three elements (name id, evr id and flags). Name ids and evr ids are offsets to previously fetched strings. In the third part are offsets to directory string names. In the next data block are keys to keeping other information contained in package header. They consist of name, type, size and storage components. All of them are stored as uint32. A key could be type of *void, constant, constantid, id, num, num32, dir, str, binary, idarray, relidarray, dirstrarray, dirnumnumarray, md5, sha1, sha256, fixarray, flexarray or deleted*. The remaining sections put schema and data together for each package.

3.6 PackageKit Use Cases

PackageKit stores information about installed or upgraded packages to Yumdb, namely to *from_repo, install_by, reason, releasEver and ReleaseVersion* sections.

Additionally PackageKit owns a separate database for classification application to groups. That is being used when the user searches for application in specific category. File is located in */var/cache/PackageKit/groups.sqlite*.

PackageKit needs for USD are basically speed requirements. All the reads need to be an order of magnitude faster than the writes, after all, PackageKit spends 99% of the transactions just reading stuff.

3.7 Gnome Software Use cases

Gnome Software application can utilize PackageKit or OSTree among others as a backend. Gnome team defined a new data file, which the upstream project can optionally translate using the same technique as GSetting schemas or Desktop files. Rather than create a new schema from scratch, it is using a subset of the AppStream metadata proposal [4]. Applications wishing to have long descriptions, screenshots and other useful things are required to ship one or more files in */usr/share/appdata/<id>.appdata.xml*. Appdata introduces the following XML tags to expand application's metadata:

- $< id/ >$ – It is the same name as the installed .desktop file for the application.
- $< metadata\_license/ >$ – Tag is indicating the content license that you are releasing the AppData text file and screenshots under.
- $< project\_license/ >$ – Tag is indicating the licenses that you used for the application and any data or media files used. This is not typically the same as the metadata license.
- $< name/ >$ – If this tag is omitted, Gnome Software collects the strings from desktop menu. In some cases it might be required to have a different name in the app-store, but most appdata.xml files will not need this.
- $< summary/ >$ – This tag could also be skipped when vendor wants to use the same summary as the one in desktop menu.

- < *description*/ > – The long description is an important part of the file.
- < *screenshots*/ > – All screenshots should have a 16:9 aspect ratio, and should have a width no smaller than 620px
- < *url*/ > – Link to the application's homepage.
- < *updatecontact*/ > – If *updatecontact* tag is included, a notification email will be sent when the standard is updated and metadata needs to be modified.
- < *project_group*/ > – The *project_group* tag identifies your project with a specific upstream umbrella project. Known values include GNOME, KDE, XFCE, MATE and LXDE.

3.8 OSTree Use Cases

With OSTree one can utilize any build system he/she likes by exporting filesystem into it on a build server. OSTree repository can be therefore exported and shared via static HTTP. Each workstation can be synchronized by running `ostree admin upgrade` command. The new content will be put into newly created root directory that will replace the old one after reboot. This provides fully atomic upgrades. Any changes made to /etc are propagated forwards, and all local state in /var is shared [3].

OSTree saves snapshots of packages installed on your system like CVS and maintains Yum configuration in JSON files as well. This tool's purpose is completely different from applications mentioned above and its database is very specific. The data stored are not single package metadata but rather a collection of files installed including UNIX permissions. It will not be considered anymore in the rest of the document.

Chapter 4

General Design Of Unified Software Database

This chapter introduces the general design scheme and API of USD that bases itself on the previous chapter about storages used by package managers nowadays. I will go through the additional requirements and features that USD should support. Last but not least I will cover the database structure and database engine candidates.

My purpose is to make a consistent storage with a convenient API that could be used easily by any package manager. Rewriting package managers to utilize USD is not part of my job. The rise of USD will extend over several epochs so that applications could accustom little by little to small changes when every release will take over a new portion of data.

The first milestone (the USD part covered in this thesis) will be merging yumdb and the history database (see 4.1). The current database holds many redundant data and is missing some critical information (reason field in transactions). The new yumdb will slightly integrate with the history database, making it possible to execute more complex queries. This way we can save the world from having another package handler. From the user's point of view, all fields from trans_data_pkg, pkgtups and yumdb will be able to be accessed in the same manner. Note: this phase is mainly focused on DNF but still considering PackageKit and Yum. The design will be as much general as possible and not bound to RPM package type.

4.1 Specifications

As has been said, USD should be one central database on the system that will be used by all package managers. It should be able to process multiple transactions from different sources of packages. Moreover, each program can utilize USD as an application-related data storage with no modification made in USD source code.

There is a list of key points (mostly from RPM) that should be kept in mind regarding the central database. Note that they contradict one another – one can't be fully accomplished without sacrificing the others.

- Primary RPM database (Packages) must remain to allow for verification of the signatures. That means allowing some redundancy. On the other hand fields not needed by majority of programs don't have to be stored in USD. They can be only referenced by *installed ID* but fully accessible from USD API.

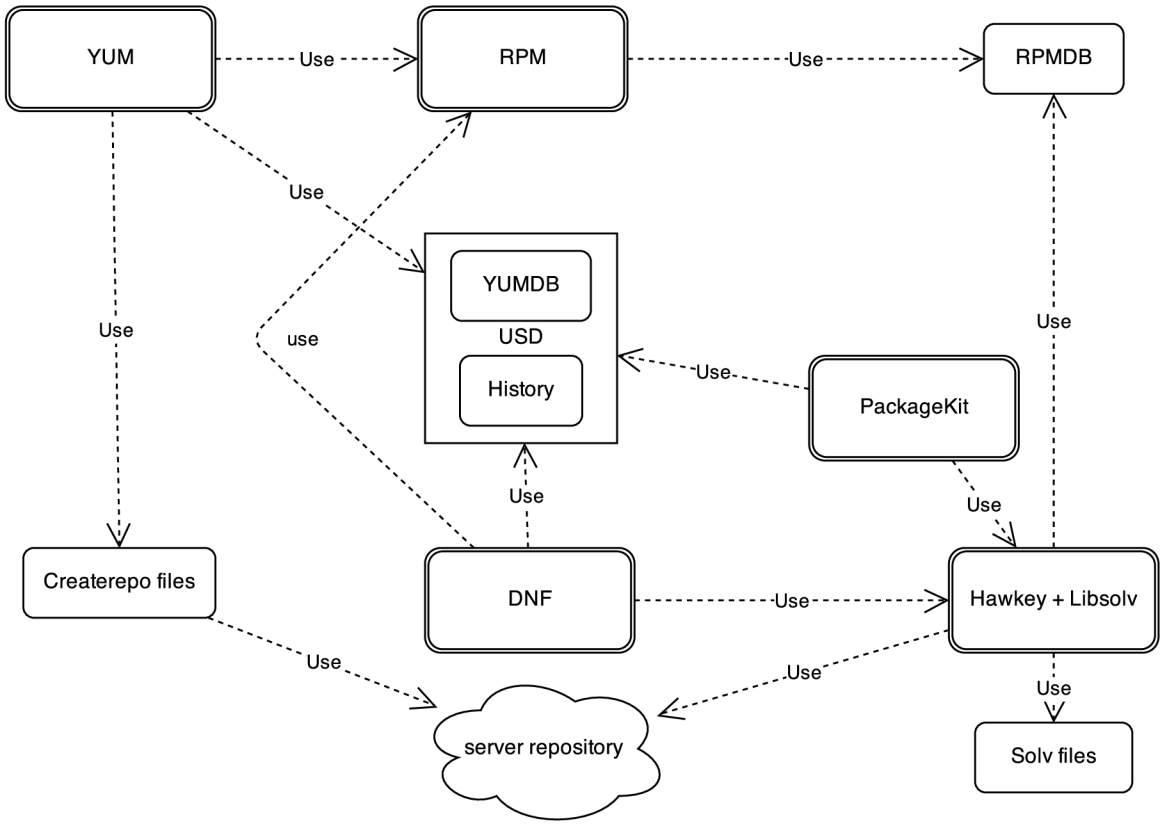


Figure 4.1: Package managers interaction between storages after the first USD's milestone

- Concurrency issues need to be considered. Traditional *one writer or many readers* scheme doesn't really cut it as people (and various applications and daemons) expect queries to work while system is being updated or software installed.
- USD has to be able to survive execution of *chroot()* command RPM uses when installing packages. External on-demand started db server process would have the benefit of being immune to chroot.
- Schema upgradability needs to be planned for the storage from start. For that reason choosing the right storage engine is critical.
- The fact that RPM is written in C and uses exact interface to RPMDB have to be kept in mind.

I will try to a find compromise between a tool for storing general packages and one more oriented for speed. There are performance demands from PackageKit. Cold cache measurements can be up to 800ms in all cases, but for starting Gnome Software this would be ideally following:

- Getting the origin (the source the package was either installed from, or last installed from) of 2500 packages: 400ms
- Getting the origin of one single package: 40ms

- Setting the origin,uid,dep/user,any-other-required-data of 200 packages: 2000ms
- Setting the origin,uid,dep/user,any-other-required-data of 1 package: 200ms
- Getting the transaction data (i.e. when installed/removed and every time the package was updated) of one package: 50ms

To avoid name conflicts when multiple applications write data to the same field and each of them expects the field to gain different types of value, the name of the field should have the prefix of the application that creates it to make the key unique.

USD can be accessible from cli. That will be used primarily for general information purpose by system administrators and package manager application developers. It will be focused on printing stats and removing necessary fields rather than querying database for a particular package. That is going to be the purpose of USD's *C/C++/Python API* that will have an object-oriented design.

As was seen in the analysis previously, package management programs tend to search record by special value and these values were indexed in SQLite files. There should be also support for creating custom indexes.

4.2 Database design scheme

There is a list of tables and fields in which the new concept differs from the current solution. The whole refactored database scheme is in picture 4.2:

- *Trans_skip_pkgs* and *trans_RPMDB_problems* records are not read or written within DNF thus can be ignored in new database design.
- *Trans_beg* and *trans_end* should be a single record that will be updated when a transaction is completed.
- *Trans_comandline* is related to the entire transaction so there is no need to duplicate it for each *trans_data_pkg*. It could be an attribute of new transaction table as their relationship is of cardinality 1:1.
- There is no need to duplicate data from yumdb to *pkg_yumdb* table. The records needed to be logged forever will be moved to transaction. The other fields will remain in yumdb only.
- *Trans_error* and *trans_script_stdout* hold the same data, they could be merged and distinguished by a type flag.
- The critical information that should stay the same (reason and group) will be moved to *trans_data_pkg*. Together with transaction records, these will remain on the system forever – as long as Yum/DNF does.
- *Pkgtup* will hold package identifier (NEVRA for RPM packages) in *name* field and *type* flag to discriminate between wide variety of package types. The other fields will be no longer used.
- *Trans_data_pkg* should store a pair of packages that are in relation instead of two separate package transaction records. We will log package transformation instead

of the exact state of a package at one moment. E.g. for upgrading and downgrading a package, the row will be ('upgraded'/'downgraded', new_package, previous_package) instead of (previous_package, 'upgrade'/'downgrade') and (new_package, 'upgraded'/'downgraded'). For 'install' or 'erase' states the package relation could be filled with information which package obsoleted it or which package required it, respectively. That way less total fields will be saved on disk when a majority of transactions are update processes.

These are rules how the tuple (state, pkg, reason, pkg) representing package transaction will be stored based on action:

- ('erased', erased_pkg, 'user', null)
- ('obsoleted', removed_pkg, 'dep', which_obsoleted)
- ('installed', installed_pkg, 'user', null)
- ('installed', installed_pkg, 'dep', which_requires)
- ('downgraded', downgraded_pkg, _, from_pkg)
- ('upgraded', upgraded_pkg, _, from_pkg)
- ('reinstalled', reinstalled_pkg, _, null)

To summarize, here are all the changes made for tables and their values in database structure:

- transaction (merged *trans_beg* and *trans_end*)
  - \+ commandline
- trans_data_pkgs
  - \+ package in relation
  - \+ reason
  - \+ group
  - − rpmdb keys
  - − commandline
- yumdb
  - − reason
  - − group_member
  - − installed_by
  - \+ rpmdb keys
- pkgtup
  - \+ type
  - − arch
  - − epoch

  - version
  - release
  - checksum
- trans_output (merged *trans_script_stdout* and *trans_error*)
  + type

– trans_skip_pkgs

– trans_cmdline

– trans_prob_pkgs

– pkg_RPMDB

– pkg_yumdb

+ repos

+ groups

Addon data

Addon data files consist of key-value pairs. The total count of keys and their key names is usually the same across all transactions. The only record regularly changed is the commandline which is stored also in the transaction itself. Addon data part is the most problematic. It is not a general solution but very specific to DNF and even the data itself are not crucial for DNF. In the meantime the file database could remain and later could be refactored to store the difference of configurations between transactions or eventually dropped.

4.3 Features

Transaction and all transaction related data (*trans_with_pkgs*, *trans_data_pkgs*, *trans_output*, *pkgtups*) can be modified only by transaction handler that created the transaction, not by a handler obtained from queries. The only data that can be changed/added from queries handler are the ones in yumdb.

Pkgtup can be dynamically expanded and by default will store all the data that were originally in yumdb (except *reason* field) plus additional info from RPMDB table. The RPMDB keys are used only for information output about packages on user demand. They can be dropped in future with no harm done.

Pkgtup can be created anytime and will be created automatically when writing additional data and no record of that package exists in the database yet.

USD could be utilized as a storage for all additional package metadata, no matter what package type it is. To avoid repeated setting of the field with package type when creating a package object or in the query filter, applications that handle only one type of package format can pass the default_type parameter to the constructor of USD to make this field set implicitly.

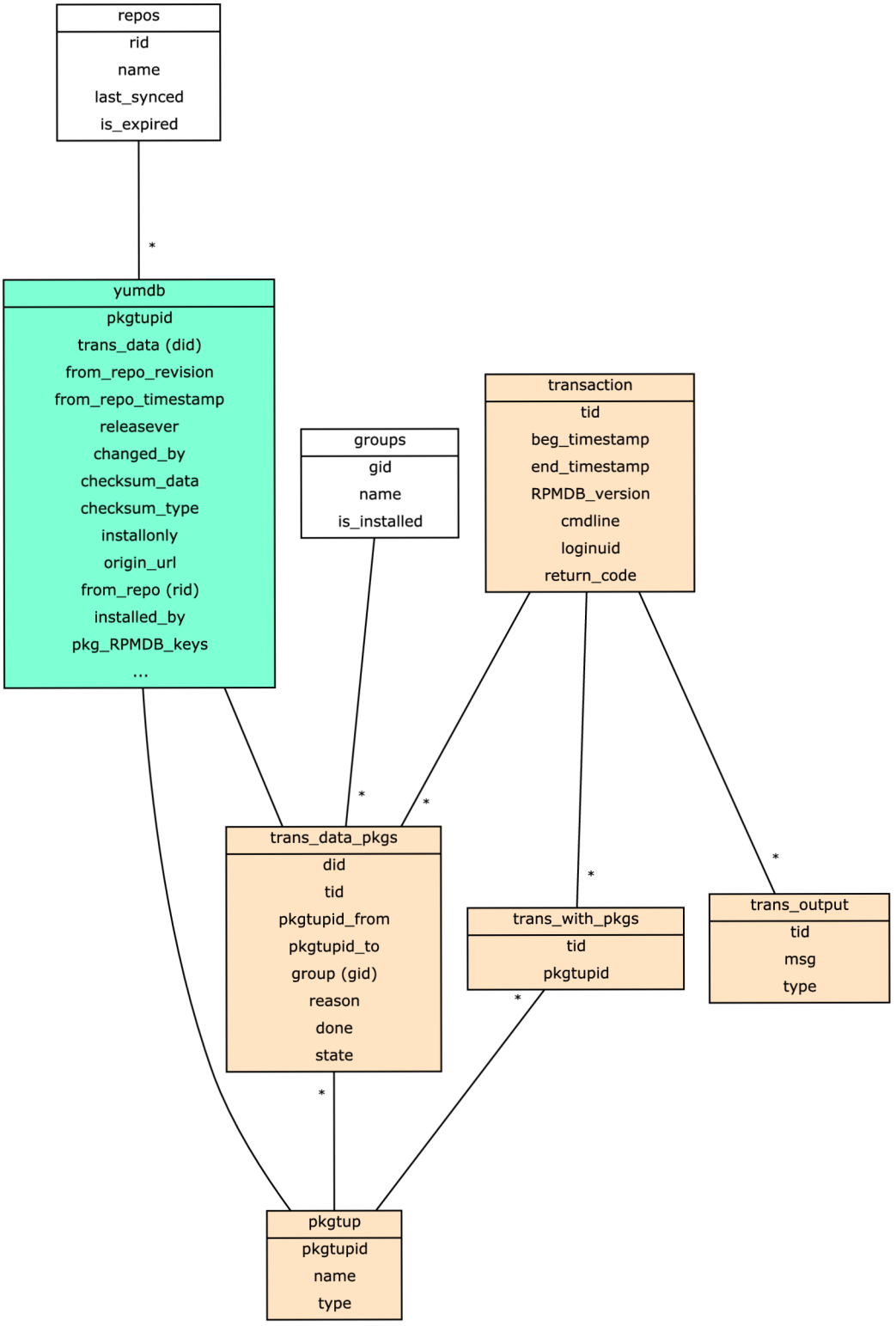


Figure 4.2: New database scheme of history database and yumdb for Yum, DNF and PackageKit

4.4 API

USD will be written in C++. Since primary user will be DNF, the API will be accessible from Python. The query syntax will be inspired by Django Query methodology [2]. The reason why we can't reuse Django's code is because it should be accessible by PackageKit as well. The Python scripts can be executed from C but this is not a clean solution that would fit into PackageKit's speed limits. From C++ code base, C bindings can be done easily.

The exact USD public C++ API with detailed description is explained below:

- `class Swdb` – handler to USD
  - `Swdb(string db_path=''var/lib/usd'', int default_pkg_type=RPM_PKG, vector<string> actors)` – constructor
    - `db_path` – from where database should be opened or where new database should be created
    - `default_pkg_type` – Search or insert records only with the same package type. If zero is set all queries will run against all kinds of packages (can be explicitly filtered in query afterwards).
    - `actors` – NEVRA strings of package managers that are responsible for following transactions.
  - `Record record(string table_name)` – Returns new Record object that could be inserted to table named *table_name* as a row after setting fields and firing *save* method.
  - `Query query(string table_name)` – Returns new Query object. Results will be from the *table_name* table only.
  - `create_index(const string& table_name, const string& field_name, bool unique=false)` – create index on frequently searched field that could have been created on demand.
    - `field_name` – column that should be indexed
    - `table_name` – table where the column is
    - `unique` – create unique (true) / normal (false) index
- `class Record` – class representing row to insert, update or delete. Object ensures database consistency whether mentioned operation is allowed or not. Object must be initialized from Swdb method.
  - `bool is_in_db()` – Returns True if the record is already in database else returns False.
  - `bool is_changed()` – Returns False if object is stored in database with the same values as it holds now, otherwise returns True.
  - `int id()` – Returns *id* of the record or -1 if item is not present in database.
  - `bool set(const string& key, int value)` – For all *append* and *set* methods stand: new fields are not pushed into database (see *save* method) and return False in the case that column type is unlike given value type, key contains non-alphanumeric character or '_id', record is trying to be updated in protected table or database connection error occurred.

  - `bool set(const string& key, const string& value)` – Sets new key with value or updates value for given key.
  - `set(const string& key, vector<Record> records)` – Insert records from vector into neighbor table having many-to-one relation to origin record table. Erase all existing connected rows in foreign tables.
  - `bool append(const string& key, Record record)` – Same as previous method except all existing records are not replaced with the new ones appended.
  - `bool get(const string& key, int& value)`
  - `bool get(const string& key, string& value)` – Return False if column in table does not exist or database error happens. Otherwise return True and assign attribute of the record to *value*.
  - `bool save()` – Returns False when database error occurs or commits all fields set or appended to database and returns True. Each appended record will implicitly call recursively *save* method itself. For every unknown key new column will be added to the table.

- `class Query` – class required for reading data from database. It includes iterator class inside that is inherited from STD input_iterator. Object must be initialized from Swdb method.
  - `filter(const string& path, string value, int value_flags)` – Sets condition of sought rows. When query is executed iterator will run through records that met all filter criteria. Any number of filters can be applied on a query.
    - `path` – Optional table names, that are in relation, ending with required column name. All the names are separated by *dot*.
    - `value` – value that should match column (last item of the path).
    - `value_flags` – should be one of EQ (value equal), NEQ (not equal), GT (greater), GTE (greater or equal), LT (less) or LTE (less or equal) comparators and any of ICASE (match case insensitive) and GLOB (match column against glob search pattern).
  - `Record operator[](int n)` – runs the query and return n-th record that succeeds on all the filters.
  - `iterator begin()` – returns iterator over Records of the query
  - `iterator end()` – end of iterator

The API of USD is objectively more general while providing more expressive power than the very specific methods in historydb. One can query literally for anything via one class and one method. The quantity of functions is replaced with fewer methods that are enhanced by parameters they take. Moreover, the core of USD will also take care that nobody overrides protected fields.

Below are USD code snippets that will be called from DNF for certain task:

- Get *reason* of the last transaction of the package. If the package reason does not exist, the next transaction will be fetched. This case is not trivial with historydb – joining tables has to be done by hand and only for tables that have implemented methods for accessing their fields.

```
usd.query("pkg_change").filter(reason__neq=None, pkg__name=<NEVRA>)[-1]
```

- Get the first transaction (`historydb.old('1')[0]`)

  ```
  usd.query("transaction")[0]
  ```

- Get the last transaction (`historydb.last()`)

  ```
  usd.query("transaction")[-1]
  ```

- Get last n transactions that contain given transaction ids (`historydb.old(tids, limit=n)`)

  ```
  usd.query("transaction").filter(tid=tids)[:n]
  ```

  Note: when the value passed to the *filter* is a List type, USD will try to match field of row to any object of the sequence.

4.5 Overview Of Storage Options

In this section I will analyze various types of database engines. I am trying to find the best storage that fits USD needs.

SQL

The storage engine for our case should be as lightweight as possible, but the majority of relational databases utilizes a server that runs on a specific port. Only a few of them are embedded and mature enough for this project, SQLite and MySQL embedded. The advantage of SQLite is the fact that it is installed on most distributions by default.

Relational attitude doesn't suit best for columns filled with data only in small amount of rows.

Security of the fields could be provided using views, although views in SQLite are read-only. That means special API should be created and deprecate SQL commands.

Key-value Store NoSQL Databases

The database is a simple data file containing records, each is a pair of a key and a value. Every key and value is serial bytes with variable length. Both binary data and character string can be used as a key and a value. Each key must be unique within a database. There is neither concept of data tables nor data types. Records are mainly organized in hash table or B+ tree. In this category belongs Berkeley DB – current RPM database backend.

- **MongoDB** is a cross-platform document-oriented database system. It eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemes.

  The features are:

  - search by field
  - regular expression searches

- MapReduce for batch processing of data

Using MongoDb as USD storage would be the most convenient solution. All custom and main fields would be retained in one object with support of powerful queries above all of them.

The downside of MongoDB is the inability to hide and protect key data from overwriting and running server is required. The biggest drawback of this storage is the absence of transactions. That would lead to inconsistency of the database [7]. Moreover it is over 90Mb of additional dependency.

The next representatives in subsections are all embedded and provide transaction operations.

- **Kyoto Cabinet**, the next USD database engine candidate, attains performance improvement in retrieval by loading the whole of the bucket array onto the RAM. The bucket array saved in a file is not read into RAM with the *read* call but directly mapped to RAM with the *mmap* call. Therefore, preparation time on connecting to a database is very short, and two or more processes can share the same memory map [11].

  The extension Kyoto Tycoon is provided for concurrent and remote connections to Kyoto Cabinet. Kyoto Tycoon is composed of the server process managing multiple databases and its access library for client applications. The server and its clients communicate with each other by HTTP RESTful interface. In addition, several operations are available in an efficient binary protocol [12].

  The features are:

  - read/write locking by records
  - storage selection: onmemory, single file, multiple files in a directory
  - MapReduce functions that can be executed parallel using multiple cores
  - prefix/regex matching

- **UnQLite** is a standard key/value store with a rich feature set [18].

  The features are:

  - support for on-disk as well in-memory databases
  - built with a powerful disk storage engine which support O(1) look-up
  - thread safe and full re-entrant
  - concurrent reader support

- **LevelDB** is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. It is mainly optimized for parallel writing [6].

  The features are:

  - Users can create a transient snapshot to get a consistent view of data.
  - Forward and backward iteration is supported over the data.
  - Data is automatically compressed using the Snappy compression library.

  - External activity (file system operations etc.) is relayed through a virtual interface so users can customize the operating system interactions.

  Unfortunately, LevelDB doesn't allow more than a single instance of the database to be open. All of the options are for a single process.

- **LMDB** is an ultra-fast, ultra-compact key-value data store utilized in OpenLDAP Project. It uses memory-mapped files, so it has the read performance of a pure in-memory database while still offering the persistence of standard disk-based databases [17].

  The features are:

  - Fully transactional, full ACID semantics
  - Readers don't block writers and writers don't block readers. Writers are fully serialized, so writes are always deadlock-free
  - Supports multi-thread and multi-process concurrency
  - Memory-mapped, allowing for zero-copy lookup and iteration
  - Provides in addition many database forks (SQLite, BDB, ...) that uses LMDB as a backend while preserving the original API.

  LMDB should be the fastest for random and sequential read of all storages mentioned above, however, it is a pretty new project that offers only commercial support.

With SQLite we would gain features like indexes and easier connection between entities for free – we won't have to reinvent a wheel. By doing so we will also attract more applications to utilize our storage thanks to the independence of any other storage library.

4.6 Implementation Details

Due to the fact that USD API is inspired by Django queries [2], user doesn't need to join tables. He/she actually doesn't manipulate SQL command strings at all. The SQL instructions are constructed internally from query path. For example, the following directive

```
Query q(db, "t3");
q.filter("t2.t1.t1f1", "val", EQ);
```

is translated into

```
SELECT t3.* FROM t3
JOIN t2 ON t3.t2 = t2._id
JOIN t1 ON t2._id = t1.t2
WHERE t1.t1f1 = 'val';
```

To make it work, table relations need to be configured during USD initialization. After query execution, the validity of the path is checked. Note: Queries are lazily evaluated. SQL commands are built from all filters applied on Query when the results are demanded, i.e. when *operator[]* or *begin* method is called. Eventually, database transaction is executed.

The implementation language of USD is C++. To create bindings to Python, so that DNF and Yum can use it, there are plenty ways of doing this. The most common way is to write a lot of redundant code in Python C API like Hawkey, librepo and libcomps does.

A more convenient solution is to declare which functions, classes and their methods should be exposed to Python and automatically wrap the original code to make it accessible from other languages. Swig is a good example of such tool that can generate bindings to dozens of languages from C/C++ code base. Another option is Boost-Python that belongs to Boost libraries family although it doesn't require any of them for runtime. Only Boost-jam (Boost configuration and build tool) is needed for a build. Boost-Python can be linked statically or dynamically, whatever is preferred. I chose dynamically linked Python-Boost in USD project to make the work of exporting interface easier.

To follow the same manner of USD's siblings (Hawkey, Libsolv, Librepo, Libcomps, DNF), the project build is managed by CMake that should check the prerequisites and generate Makefile.

Chapter 5

USD Testing

This chapter is focused on benchmarking of USD and yumdb.

5.1 Policy Of Measurements

In test cases I am comparing yumdb and USD part for storing direct package metadata. Since historydb and USD both utilize SQLite as a backend, it is pointless to perform a comparative analysis.

The test scripts were written in C++ and measured with `chrono` library from the all-embracing C++11 standard. The procedure of benchmarking was following:

- List of installed packages' NEVRA names on my system is obtained by executing `rpm -qi` command, then records were shuffled and first n names temporary stored.
- The names are appended into vector
- Timer is set
- Loop for querying/writing items from/into USD/yumdb is fired.
- Calculation the execution time (user) of the loop
- This procedure was repeated ten times from step three and minimal time was counted as final result.

Every operation will be tested for 50, 200 and 3500 records for both USD and yumdb. With regard to chosen numbers, the fifty packages set is the smallest set providing relatively accurate measurements. This is the average package count when installing new software that has a lot of dependencies out of Fedora's pre-installed package set. 200 records will be queried for a match in USD usually during update that is done on weekly basis. The largest number is the total number of all installed packages on my system during its lifetime. Note that in real, packages will be filtered by Libsolv at first then put into USD query for extra metadata information.

Benchmarks should figure out whether PackageKit performance requirements are satisfied. Yumdb is tested just out of curiosity and it doesn't really compete with USD, only to give us some perspective of how long the operation usually takes. Speed is not USD's top priority as long as it is not super slow.

At the end of each test case is a sheet with measured values. In Summary Of Test Cases section 5.6 are these measures in graphs, collected and aggregated by number of packages the benchmark was run on.

All tests were conducted on a machine with 7.5GB RAM, Intel Core i7-3520M 2.9Ghz per core, 4 cores total, hard drive of 7200 rpm. Filesystem had ext4 partitioning and sqlite3 of version 3.8.3 was installed.

5.2 Read Performance

The database was filled with 3500 NEVRA records as name and for each, the reason of package installation was additionally supplied that was constant for all of them – 'user'.

50, 200 and 3500 random NEVRA samples were randomly searched by name for reason tag within USD and Yumdb. The process was applied to warm cache and cold cache separately. Before every cold cache read measurement, this command was executed to free both page cache and dentries cache.

```
sync ; echo 3 | sudo tee /proc/sys/vm/drop_caches
```

Results can be seen in the next two tables. According to the numbers YumDb outperforms USD even with indexed name column. Indexed USD was about eight times faster that its non-indexed version only for read of 3500 records.

number of packages \database	USD (indexed)	USD	yumdb
50	0.003	0.002	0.001
200	0.008	0.011	0.001
3500	0.139	1.048	0.017

Table 5.1: The execution time of read (warm cache) operation in sec.

number of packages \database	USD (indexed)	USD	yumdb
50	0.005	0.005	0.004
200	0.009	0.009	0.018
3500	0.153	1.066	0.076

Table 5.2: The execution time of read (cold cache) operation in sec.

5.3 Write Performance

This performance test should be analogous to setting metadata for packages which were verified after installation. The insertion of 50, 200 and 3500 records were measured.

In the table 5.3 the time difference between storages is significant. In Yumdb time sheet, the results are linear to number of records whereas the trend in USD is likely logarithmic.

number of packages \database	USD	yumdb
50	0.002	0.011
200	0.002	0.022
3500	0.006	0.239

Table 5.3: The execution time of write operation in sec.

5.4 Erase Performance

The whole USD and yumdb environment made in previous test case for 50, 200 and 3500 records was timed for complete database purge.

The deleting of USD took almost zero milliseconds in any case while in yumdb for 3500 items it lasted 30 milliseconds.

number of packages \database	USD	yumdb
50	0	0
200	0	0.001
3500	0	0.033

Table 5.4: The execution time of erase operation in sec.

5.5 Disk Space Taken

In this section the amount of disk space USD and yumdb takes for 50, 200 and 3500 items is examined. The capacity of storages from write performance check 5.3 is measured in bytes.

As it may seems on first impression that yumdb is wasteful in sense of storing values in separate files, from tables 5.5 can be seen that SQLite takes up even more space.

number of packages \database	USD	yumdb
50	12288	416
200	19456	1708
3500	178176	28320

Table 5.5: The total disk space taken by databases in bytes

5.6 Summary Of Test Cases

The performance of databases is more or less at the same speed as for cold cache reads. Yumdb is better in warm cache read operations while taking less disk space. On the other hand, USD is faster in inserting records into database and their deleting.

From statistically collected measurements it was ascertained that USD's execution for read procedure is not blazingly fast but still sufficient. That could be eventually an argument for Package Managers to abandon USD. In that case some optimizations need to be considered.

When USD is ready for production, profiling PackageKit and DNF for install, removal and update transactions targeting on database part will be essential.

These graphs are collected from previous tests to provide better visual comparison.

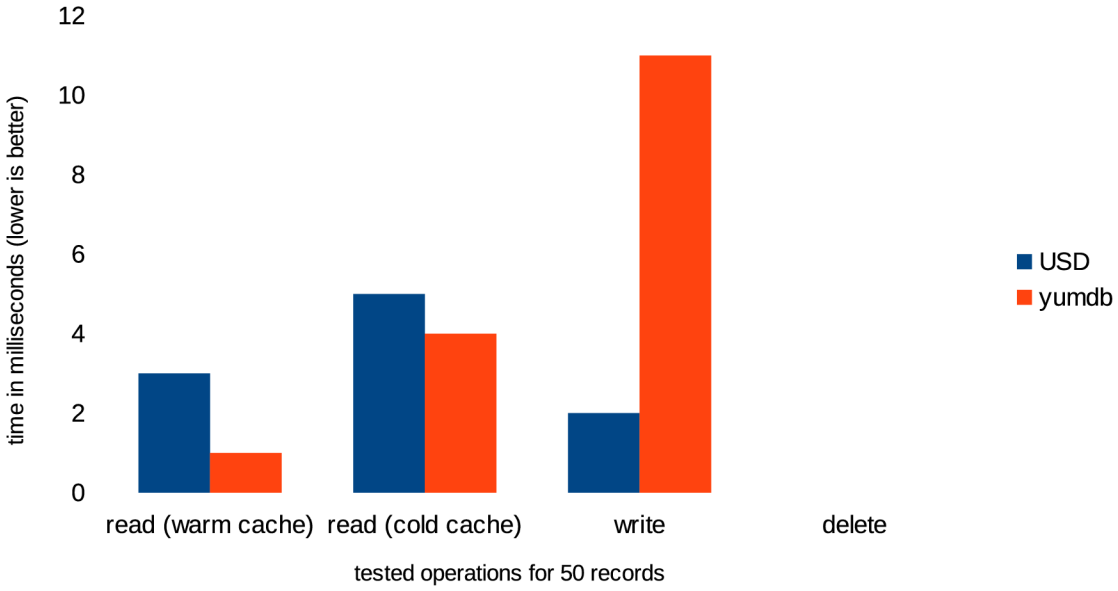


Figure 5.1: Benchmarks for 50 records

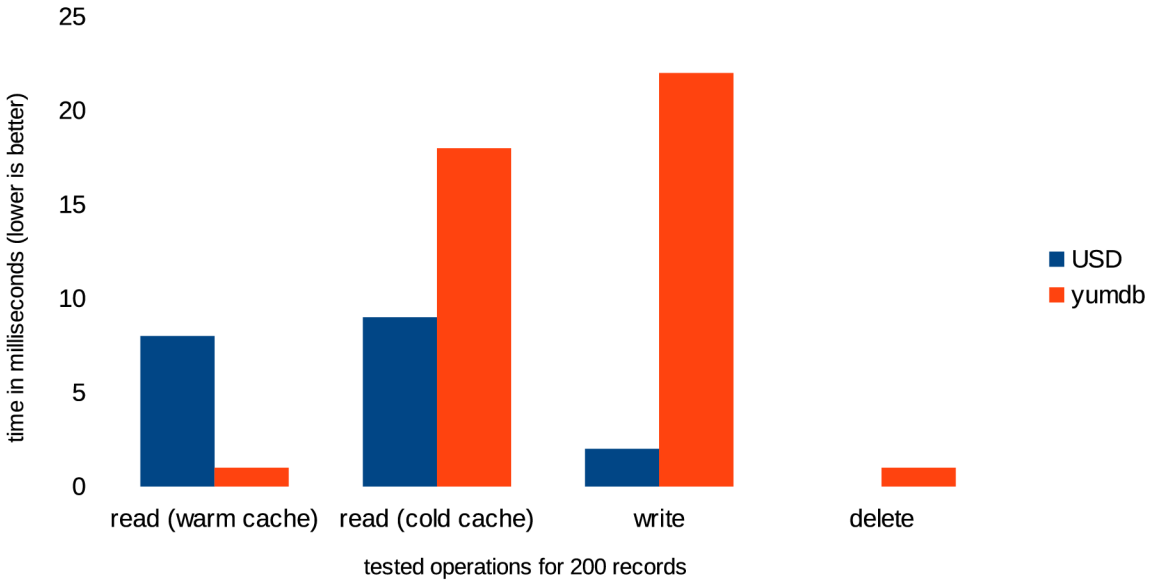


Figure 5.2: Benchmarks for 200 records

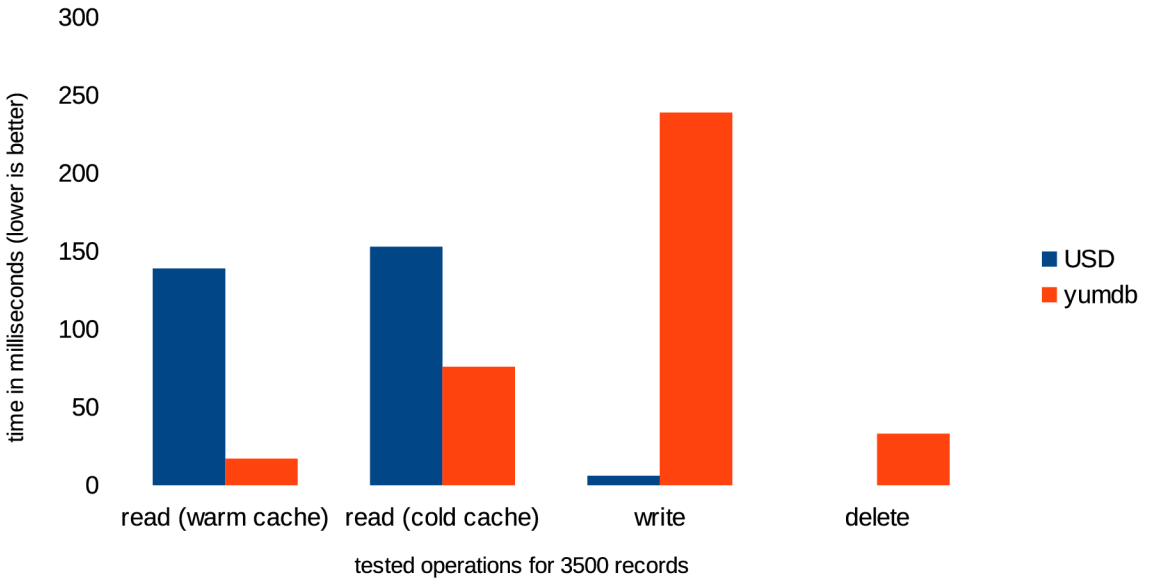


Figure 5.3: Benchmarks for 3500 records

Chapter 6

Summary

This chapter should summarize the thesis. What was done and what could be done better will be discussed. I will go through the future of USD as well.

6.1 The Future Of USD

Conversion from yumdb and historydb to USD is in DNF's roadmap and its absence blocks several issues from being solved. Pull request of USD for DNF was created. Now some adjustments and more test cases from DNF should be added to USD. The plan is that Fedora main repository will host *usd*, *python-usd* and *usd-devel* packages to enlarge the package management ecosystem.

After a successful era of trouble-free functioning of USD as DNF's main storage, more focus on C part will be made and then PackageKit should start using it.

Yum should use USD at the same time as PackageKit. There is a possibility that Yum will not adapt to the new trend at all as it is marked as obsolete.

In the next phase, either a thin API should be created above USD and RPM or the access to RPM transactions from within USD should be implemented to eliminate occurrence of RPM transaction that is not logged into transaction history.

The last move will be refactoring RPMDB. BDB's license is changing and new data storage should replace the current solution sooner or later. The final solution of USD is represented in picture 6.1. It would be great as a general solution for all kinds of packages and it would save Package Managers a lot of trouble for taking care of database but will be hardly deployed. It would replace RPMDB, createrepo files, history database and yumdb at once. This ambitious plan would require a huge portion of changes in RPM code base. In appendix A is a preview what API should contain to be package-type agnostic. The user should be able to operate with five base classes: Application, Relation, Repo, File and Directory. These classes store commonly used attributes across package types. Any more specific category defining the new fields has to be derived from the base class. By doing this we can manipulate different data sources uniformly.

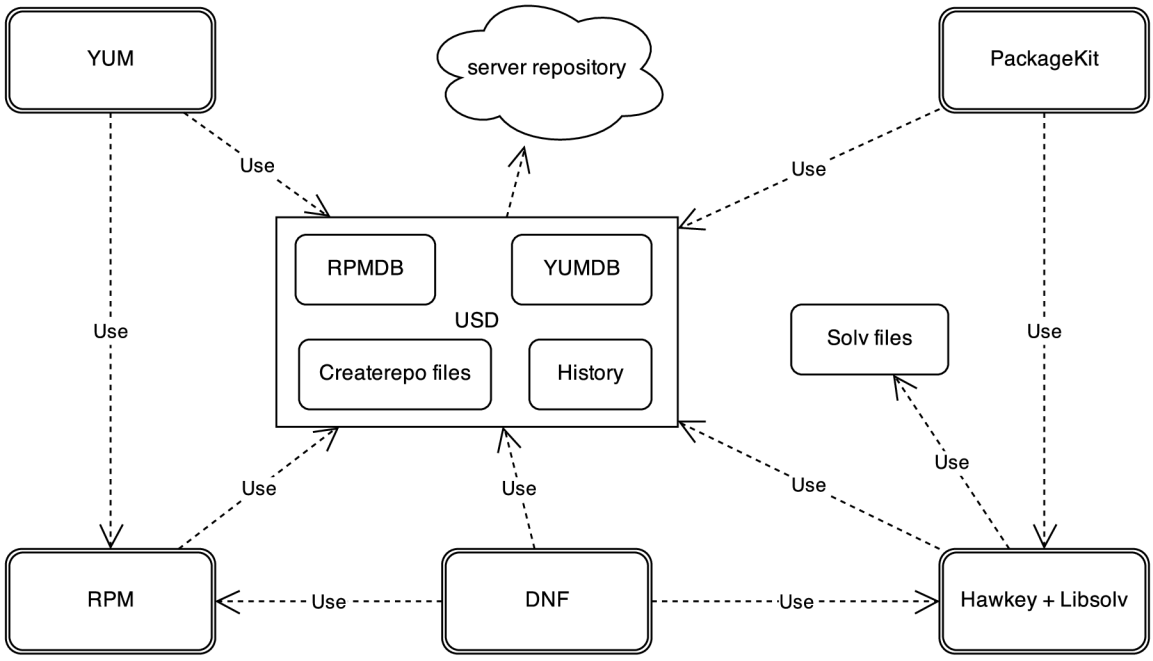


Figure 6.1: Ideal package managers interaction using USD

6.2 Possible Improvements

I can focus on optimizations, especially speeding up the read operation. Some improvement could be accomplished by switching from SQLite to SQLightning which utilizes LMDB as storage engine. The creators of SQLightning claim that it's three times faster than SQLite3 for random reads. Aside from performance part, a little bit of API changes based on sane arguments from package managers will be inevitable. Only time will tell which. However, the idea of logging all package changes to USD, database structure and base API for queries will remain in any case.

6.3 Conclusion

Although USD has deflected from the original plan of storing all package metadata and replace RPMDB, its current implementation is still necessitous for high-level package managers. The importance of this thesis consisted in determining a usable API to universal extensible storage. USD maintains package transactions logs; package, repository and groups additional data that couldn't be stored in RPMDB. The core package data are still maintained by RPM. Hopefully, the replacement of RPMDB and unifying with USD will arise.

It will take a while before it is adopted by most package managers. I will write a 'how to start with USD' document and optionally make patches for other package management tools, besides DNF, to ease the translation.

Aside from DNF, Yum, PackageKit, OSTRee and Gnome Software; some release engineers could be interested in USD for their projects. USD is package-type agnostic and can be utilized for instance on Debian based operating system. I am open to adding features

according to additional sudden requirements as well.

USD has ACID operations and its performance is enough for brief start of Gnome Software and PackageKit applications. DNF cares rather about API convenience and getting rid of ancient Yum's code.

To Summarize, USD accomplished goals it was created with and it will be ready for production once patch proposal is accepted by DNF, then it will be marked as DNF's dependency and added to Fedora main repository.

Bibliography

[1] Edward C. Bailey. *Maximum RPM*. Red Hat Inc., 2600 Meridian Parkway, Durham, NC 27709, 2011.

[2] Django Software Foundation. Django documentation. https://docs.djangoproject.com/en/dev/topics/db/queries/, 2013. [Online].

[3] Gnome. Ostree. https://wiki.gnome.org/Projects/OSTree, 2014. [Online].

[4] Richard Hudges. Gnome software. https://wiki.gnome.org/Apps/Software, 2013. [Online].

[5] Richard Hudges. Packagekit. http://www.packagekit.org/pk-intro.html, 2013. [Online].

[6] Google Inc. Leveldb. https://code.google.com/p/leveldb/, 2013. [Online].

[7] MongoDB Inc. Mongodb manual. http://docs.mongodb.org/manual/faq/fundamentals/, 2013. [Online].

[8] Red Hat Inc. Yumdb - yum. http://yum.baseurl.org/wiki/YumDB, 2012. [Online].

[9] Red Hat Inc. Rpm package file structure. http://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/ch-package-structure.html, 2013. [Online].

[10] Red Hat Inc. Software collections. https://fedorahosted.org/SoftwareCollections/, 2013. [Online].

[11] FAL Labs. Kyoto cabinet. http://fallabs.com/kyotocabinet/, 2013. [Online].

[12] FAL Labs. Kyoto tycoon. http://fallabs.com/kyototycoon/, 2013. [Online].

[13] Jan Nový. Beer. http://sourceforge.net/projects/beer/, 2014. [Online].

[14] Fedora Project. Fedora project. http://fedoraproject.org/wiki/PackagingDrafts/Epoch, 2009. [Online].

[15] The Chromium Projects. Chromium os autoupdate. http://www.chromium.org/chromium-os/chromiumos-design-docs/autoupdate-details, 2014. [Online].

[16] Michael Schröder. Libzypp satsolver. http://en.opensuse.org/openSUSE:Libzypp_satsolver, 201. [Online].

[17] Symas. Lmdb. http://symas.com/mdb/, 2013. [Online].

[18] Symisc Systems. Unqlite. http://unqlite.org/, 2013. [Online].

[19] Jan Zelený. Design of new rpm database. Master's thesis, FIT VUT v Brně, 2010.

Appendix A

USD Future C++ API

- `class Db` – Db is the main class for access to the database.
  `public:`
  - `Db(string& namespace)` – Namespace is typically program's name that is calling the constructor.
  - `bool close()` – Close the database and commit all changes. Return false in case of error.
  - `void begin_transaction(...)` – Anything written after this line will not be stored into database till end_transaction() will be executed. Parameters are strings of namespace where the fields will be modified.
  - `void end_transaction()` – Commits changes that were made between this line and the begin_transaction() command.
  - `void discard_transaction()` – Discard changes that were made between this line and the begin_transaction() command.
  - `int last_synced(app_type type)` – Return time in UTC of last synchronization with master database or remote repositories. For example if type is RPM_PACKAGE, it will return time of last checking the RPMDB and repositories with librepo.
  - `bool force_sync(app_type type)` – Synchronize immediately with master database of given type. Return false if an error occurred (e.g. no internet connection to fetch remote repository).
  - `Entity<Application> apps()`
  - `Entity<File> files()`
  - `Entity<Directory> dirs()`
  - `Entity<Repo> repos()`
- `class Entity` – Entity class represents base classes.
  `public:`
  - `Entity(string& file)` – Argument is path to file.
  - `int last_modified()` – Return UTC of last modification. Application that make own cache database from USD can take advantage of this.
  - `int size()` – Return number of stored records.

  - `Query<T> query()`
  - `bool create_index(string key)`
  - `bool delete_index(string key)`

- `class Query : public std::iterator<std::input_iterator_tag, T>` – Query is an iterator that can access sequentially to all records that satisfy the condition.
  `public:`
  - `Query(Condition)`
  - `Query(const Query&)`
  - `Query& operator++()`
  - `Query operator++(int)`
  - `bool operator==(const Query& rhs)`
  - `bool operator!=(const Query& rhs)`
  - `T& operator*()` – Returns the record of certain class.

- `class Condition` – This class is used for merging two Conditions together.
  `public:`
  - `operator&(Condition)`
  - `operator|(Condition)`
  - `and(Condition)`
  - `or(Condition)`

- `class Match : public Condition` – Match is used for searching records whose specific key equals certain value.
  `public:`
  - `Match(string key, int val)`
  - `Match(string key, string val)`
  - `Match(base_field field, int val)`
  - `Match(base_field key, string val)`

- `class NotMatch : public Condition` – NotMatch is used for searching records whose specific key not equals certain value.
  `public:`
  - `NotMatch(string key, int val)`
  - `NotMatch(string key, string val)`
  - `NotMatch(base_field field, int val)`
  - `NotMatch(base_field key, string val)`

- `class CustomData` – It is an interface that allow write custom data to any base class.
  `public:`

- `void set_custom(string& key, int value, override = true)` – Set int value with given key into implicit namespace (Db class constructor parameter). If override is set to false and some value with given key exists then value remains unchanged.
- `void set_custom(string& key, string& value, override = true)` – Set string value with given key into implicit namespace (Db class constructor parameter). If override is set to false and some value with given key exists then value remains unchanged.
- `void set_custom(string& namespace, string& key, int value, override = true)` – Set int value with given key into explicit namespace. If override is set to false and some value with given key exists then value remains unchanged.
- `void set_custom(string& namespace, string& key, string& value, override = true)` – Set string value with given key into explicit namespace. If override is set to false and some value with given key exists then value remains unchanged.
- `bool get_custom(const string& key, int& value)` – Fill value and return true if key in implicit namespace exists, otherwise return false.
- `bool get_custom(const string& key, string& value)`
- `bool get_custom(const string& namespace, const string& key, int& value)` – Fill value and return true if key in namespace exists, otherwise return false.
- `bool get_custom(const string& namespace, const string& key, string& value)`
- `void remove_custom_int(const string& key)` – Remove int value of given key from implicit namespace.
- `void remove_custom_string(string& key)` – Remove string value of given key from implicit namespace.

- `class Application : public CustomData` – Base class Application is prototype for all types of packages.
  `public:`
  - `int type()` – type of application, E.g. RPM_PKG, DOCKER_IMG, ...
  - `string name()`
  - `string version()`
  - `string description()`
  - `string group()`
  - `int version_cmp(string version)`
  - `int size()`
  - `int state()` – can be INSTALLED, NOT_INSTALLED, INSTALLING, ...
  - `string arch()`
  - `string author()`
  - `string license()`

  - `vector<File> files()`
  - `vector<Directory> dirs()`
  - `vector<AppRel> provides()`
  - `vector<AppRel> requires()`
  - `vector<AppRel> conflicts()`
  - `vector<Repo> repos()`
  - `string uninstall_script()` – uninstall command of application

- `class RpmPackage : public Application` – RpmPackage is a specific package derived from base class Application.
  `public:`
  - `int transaction_id()` – id of package set it was installed with
  - `int install_id()`
  - `vector<AppRel> obsoletes()`
  - `string release()`
  - `int epoch()`
  - `string url()`
  - `string summary()`
  - `string signature()`
  - `int evr_cmp(RpmPackage other)` – Compares epoch, version and release with other package. Returns 0, -1, 1 if this package is equal, less or greater than other package respectively.
  - operations with headers that will be wrapped above RPM

- `class AppRel` – AppRel is base class that represents dependency between packages.
  `public:`
  - `int type()` – type of AppRel (probably never needed, reserved for potential future usage)
  - `Application latest_same_arch()` – Get latest application of all application in relation that fulfill type, what, version and arch condition. Architecture of returned record will be same as user's machine processor.
  - `Application latest()` – Get latest application of all applications in relation that fulfill type, what, version and arch condition.
  - `int comparator()` – Returns LESS, LESS_EQUAL, EQUAL, GREATER_EQUAL or GREATER.
  - `int flags()` – type of dependency, etc.
  - `string what()` – name of application in relation
  - `string version()`
  - `string arch()`

- `class Repo : public CustomData` – Repo is base class representing repository from where the application can be downloaded.
  `public:`

  - `int type()`
  - `string url()`
  - `string name()`

- `class Directory : public CustomData` – Base class holds information of application's directory.
  `public:`
  - `int type()`
  - `string name()`
  - `int attributes()` – directory ownership
  - `vector<File> files()`
  - `vector<Application> application()` – Returns all applications that share this directory.

- `class File : public CustomData` – Base class represents information of application's file.
  `public:`
  - `int type()`
  - `string name()`
  - `int attributes()` – file ownership
  - `Directory dir()`
  - `vector<Application> application()` – Returns all applications that share this file.