The Ultimate Debian Database: Consolidating Bazaar Metadata for Quality Assurance and Data Mining

Lucas Nussbaum
LORIA / Nancy-Université
Nancy, France
Email: lucas.nussbaum@loria.fr

Stefano Zacchiroli
Université Paris Diderot, PPS
UMR 7126, Paris, France
Email: zack@pps.jussieu.fr

***Abstract*—FLOSS distributions like RedHat and Ubuntu require a lot more complex infrastructures than most other FLOSS projects. In the case of community-driven distributions like Debian, the development of such an infrastructure is often not very organized, leading to new data sources being added in an impromptu manner while hackers set up new services that gain acceptance in the community. Mixing and matching data is then harder than should be, albeit being badly needed for Quality Assurance and data mining. Massive refactoring and integration is not a viable solution either, due to the constraints imposed by the bazaar development model.**

This paper presents the Ultimate Debian Database (UDD),[1] which is the countermeasure adopted by the Debian project to the above "data hell". UDD gathers data from various data sources into a single, central SQL database, turning Quality Assurance needs that could not be easily implemented before into simple SQL queries. The paper also discusses the customs that have contributed to the data hell, the lessons learnt while designing UDD, and its applications and potentialities for data mining on FLOSS distributions.

***Keywords*-open source; distribution; data warehouse; quality assurance; data mining**

I. Introduction

Most FLOSS (Free/Libre Open Source Software) projects require some infrastructure: a version control system (VCS) to store source code, a bug tracker to organize development work, mailing lists for user and developer communication, and a homepage for "marketing" purposes. The needs are usually that simple, and are also similar across projects. That similarity has paved the way to the diffusion of *forge* software and providers of one-fits-all FLOSS-infrastructures, that have been steadily growing for the past decade [1]. Such a *de facto* standardization of infrastructure eases all tasks that require mining and comparing facts about, for instance, all projects hosted on the very same (or across very similar) forge(s) (e.g. [2], [3]).

While infrastructure standardization is common among FLOSS *development* projects—whose aim is developing and periodically release software components, usually in source form—, it is not as common among FLOSS *distributions*. The role of such distributions (e.g. Debian, Fedora, Ubuntu, openSUSE, Gentoo, to name just a few) is to mediate between "*upstream*" development projects and their final users. In doing so distributions try to ease software installation and configuration, often by the mean of the *package* abstractions [4]. The peculiar role of distributions makes their infrastructure needs different from those of development projects; some of their peculiar needs are:

- *archive management* that enable developers to release new packages, usually to fix bugs or in response to new upstream releases;
- *build bots* (or *build daemons*) that rebuild packages on architectures other than those available to individual maintainers;
- support for *distribution-wide rebuilds* to evaluate the impact of core toolchain (e.g. `gcc`) changes [5];
- multi-package *bug tracking* able to scale up to millions of bugs;
- *automated testing* to check that packages adhere to some distribution policy, that might change over time;
- *developer tracking* to quickly identify developers whose interest fade away [6];
- support for different *line of developments* (or *suites*, e.g.: stable, unstable, ...), corresponding to independently evolving sets of packages;

The sheer amount of packages to be managed, the resources needed, the relatively small number of FLOSS distributions, and the existence of distribution-specific customizations of the above processes have hindered the standardization of distribution infrastructures: every distribution has developed its own infrastructure. Still, the ability to combine together information coming from different distribution data sources is compelling for distributions as it is for researchers willing to mine and study facts about them.

The need of researchers to have convenient access to distribution data is easily explained: they want to focus on data mining rather than on the gazillion of technologies under which they are hidden. As an example of distribution needs to combine data, we consider *Quality Assurance (QA)*

Stefano Zacchiroli is partially supported by the European Community FP7, MANCOOSI project, grant agreement n. 214898

Unless otherwise stated, cited URLs have been retrieved on 12/01/2010.

[1] http://udd.debian.org

Table I
DESIGN CHOICES IN THE DEBIAN INFRASTRUCTURE: PROGRAMMING LANGUAGE AND DATA FORMAT

Purpose	Name	Implementation language and data storage format
Archive management	`dak`	Python, problem-specific PostgreSQL database. Data exported as text files formatted similarly to (but not equal to) RFC 2822 email.
Distribution of package builds over build daemons	`wanna-build` buildd.debian.org	Perl, PostgreSQL database (now), Berkeley DB (until recently). Data exported on the Web, with an interface not suitable for massive retrieval.
Bug tracker	`debbugs` bugs.debian.org	Perl, per-bug text files and mailbox Data exported on the Web; SOAP API available, but not suitable for massive retrieval.
Management of developer accounts	Debian LDAP keyring.debian.org	LDAP (accessible using standard LDAP tools and APIs), GnuPG keyrings.
Developers identity tracking (email addresses)	`carnivore`	Perl, Berkeley DB.
History of packages upload	—	Mailing list archives.
Upstream version tracking	`DEHS` dehs.alioth.debian.org	PHP, PostgreSQL database
Package popularity	popcon.debian.org	Perl, all results exported as a line-oriented text database, emails (to record the submissions).
Policy conformance check (à-la continuous integration)	lintian.debian.org	Perl, all results exported on the web as a line-oriented text database.
Package classification	`debtags`	C++, all data exported on the web as a line-oriented text database.
Package tracking system	packages.qa.debian.org	Python, XSLT, per-package XML files. SOAP API available, but not suitable for massive retrieval.
Web site	www.debian.org	WML (Website META Language), static HTML pages.

in the context of Debian, a very large distribution currently formed by about 27,000 binary packages [7]. The task boils down to identify sub-standard quality packages which are in need of specific actions such as technical changes, assignment to a new maintainer, or removal from the archive.

Example 1: The authors, as members of the Debian QA team, often face the following question:[2]

> *Which release critical bugs affect source packages in the testing suite, sorted by decreasing package popularity?*

The affected packages are in urgent need of attention, given how popular and at the same time how buggy they are. Implementing a monitor that periodically answers the above question turns out to be surprisingly difficult in Debian, due to the amount of involved data sources (binary and source package metadata, bug tracker, suite status, popularity, . . .) and, more importantly, due to their heterogeneity.

This paper reports about the countermeasure that has been adopted to such a difficulty—called the *Ultimate Debian Database (UDD)*—and how it implements data integration for the purposes of both data mining and Quality Assurance.

Paper structure: Section II discusses the data integration problem in Debian and the peculiar factors which have originated it, while Section III details the architecture of UDD. Section IV gives an overview of the current status of UDD and some of its figures. Some applications are presented in Section V, and a comparison with related work is made in Section VI.

[2]A few Debian terminology explanations are in order: *release critical* (RC) are those bugs whose severity makes a package unsuitable for release; *testing* is the suite meant to become the next stable release.

II. THE DEBIAN DATA HELL

The difficulty of joining together different data sources, which has plagued the Debian project for several years, is colloquially referred to as the *Debian data hell*. Before analyzing UDD as a solution to that, we summarize here the factors that have *caused* the Debian data hell:

- heterogeneity
- community inertia
- long history
- maintainers' programming and design skills
- tight-coupling *and* service distribution

Discussing these factors highlights how the data hell cannot easily appear in distributions whose infrastructures are run by (wise) companies, and how it is likely to plague only purely community-driven distributions. Among mainstream distributions, that boils down to Debian itself and Gentoo.[3]

Heterogeneity: The basic cause of the Debian data hell is of course *heterogeneity*. Even though the basic entities are fairly clear albeit complex (packages, bugs, . . .), the services dealing with them have been implemented according to sharply different design choices. To give an idea of that, Table I shows the core parts of the Debian infrastructure summarizing, for each service, the main design choices in term of implementation language and data storage format: languages vary from system-level languages such as C to scripting languages (Shell, Perl, Python) and even XSLT, data formats vary from databases of all kinds (SQL, BerkeleyDB, LDAP) to ad-hoc textual languages and even plain mailboxes(!).

[3]Private communication with Gentoo developers has indeed confirmed that they suffer, at present, of similar problems.

Community inertia: Abstracting over that heterogeneity is a typical integration problem [8]. What is peculiar is the context and that in turn impacts on the viable solutions. For instance, one cannot decide from scratch to switch all the data storage to a specific database, for several reasons:

- There is no central authority that can take the decision, it must be taken by the community. In particular, the decision must be agreed upon by the current maintainers of the affected services. It would be otherwise hard to complete the switch, and hijacking of such important infrastructure maintenance is unlikely.
- Those maintainers are often fond of their original design choices (they made them because they liked them) and not necessarily care about the "greater good" of data integration. This is neither malevolent nor surprising, it goes along with the bazaar model where hackers mainly scratch their own itches [9].
- Even the developer community at large often lacks the vision of the greater good, because data hell drawbacks are mostly experienced by a few cross-cutting teams (such as QA), and researchers which have hard times mining data. That contributes to make even harder finding consensus (and manpower) around infrastructure-wide changes: *if it is not broken, don't fix it*.

We observe that among distribution developers, the most common "itches" are packaging-related, not maintaining the infrastructure needed to achieve that. That fully explains the choice made by companies backing up distributions (e.g. Suse for openSUSE, RedHat for Fedora, Canonical for Ubuntu, etc.) of maintaining the distribution infrastructure. That enables distribution developers to concentrate their efforts on other tasks than maintaining the infrastructure. In those contexts, community inertia (for what concerns the infrastructure) vanishes: the company can decide—and then pay to achieve it—to refactor massively services to obtain uniformity and ease data integration.

Long history: Having now passed 15 years of history, Debian is one of the oldest distribution. New components of its infrastructure have been added periodically along its lifetime, as soon as someone implemented them and they gained *acceptance* among developers. The technology chosen to implement services resent from the hacker trends at the time of the first implementation. For instance, it is clear now that XML technologies are not particularly welcome among Debian developers (mostly because they are seen as bloated), but they were "trendy" when the Package Tracking System (PTS)[4] has been first implemented in 2002. That has implied that only a few people are able/willing to hack on the PTS nowadays, and inertia hinders its re-engineering (if it is not broken ...). Another peculiar example is the Debian website, which is now stuck with a technology (WML) which has been abandoned upstream.

[4] http://packages.qa.debian.org

Conversely, the long history of Debian contributes to its interest as a subject for observation [10], [11], [12]: UDD is meant to ease that, trying to compensate the drawbacks that a long history has induced.

Maintainers' programming and design skills: Members of the Debian community are good packagers and debuggers, but not necessarily good software developers or architects. When it occurs to them to create a new infrastructure component, they can made bad design choices due to their lack of experience. Later on, inertia makes hard to fix them once the service gets acceptance.

Tight-coupling and *service distribution:* The lack of a company providing the infrastructure means that hardware resources are sought via sponsoring and are scattered around the planet, where sponsors can host them. Services are nevertheless tightly-coupled and at the same time open, making their data publicly available to project developers. Hence inter-service dependencies are neither known a priori by the service maintainer, nor easily identifiable. This factor augments the resilience to change, because the community grows accustomed to the experience that any change, no matter how small, *will* break something.

Summarizing the discussed factors we obtain a rather depressing scenario for both Quality Assurance investigations and data mining, which both need an easy way to correlate distribution data sources together. UDD, which we describe next, provides a solution to that, still playing by the rules of a purely community-driven distribution.

III. Architecture

The development of the Ultimate Debian Database (UDD) started in 2008 with the aim of providing a unique solution to the data hell problem, by creating a central database where the data located in the various Debian sources would be imported. It was decided early on that such database would only act as a *replica* of the canonical data, and would not attempt to replace them: it was not considered possible at the time to win over the discussed community inertia.

A. Goals and Scope

UDD was designed and developed with three different kinds of target applications in mind: quality assurance, collaboration with derivatives, and data mining.

Quality Assurance: The Debian QA team works to improve the quality of the distribution as a whole. That includes spotting poor-quality, buggy, or unmaintained packages among the ≈15,000 source packages in Debian, and maintainers that neglect their packages among the ≈1,000 official Debian developers and the tens of thousands contributors. Such tasks require to combine different data sources about packages quality (bugs, result of automated tests, package popularity, ...) and maintainers activity (uploads, activity on mailing lists, ...). Before UDD, the identification

of neglected packages (and neglecting maintainers) was relying mainly on manual reporting by users or other developers and a few semi-automated tools [13] (for maintainers).

Collaboration with derivatives: Debian plays an important role as the root for many *derivative distributions*, including Ubuntu, Sidux, and the educational distribution Skolelinux. Collaborating with those other distributions requires the exchange of information about packages and bugs to be able to efficiently share development efforts. For instance, when a package is modified in Ubuntu to fix a bug reported by an Ubuntu user, it is important that the Debian maintainer is aware of the change to be able to integrate the change in the Debian package, and therefore keep the delta between Ubuntu and Debian at a manageable level. The use of proprietary or difficult to access data storage on both sides limits the opportunities for data exchange. Coincidentally, before UDD the exchange of information among the two distributions was limited to the respective lists of packages and their version numbers.

Data mining: As the largest FLOSS distribution, as one of the oldest, and as a purely community-driven distribution, Debian provides many opportunities for data-mining of interesting facts. However, it is often discouraging to mine Debian because of the efforts required to overcome the various problems described in section II. Since UDD has been developed by Debian developers with a lot of Debian expertise, it was possible to design a database schema which avoids the usual traps that people encounter when data-mining Debian, like the use of the same word – *package* – for different objects – *source package*, *binary package*. As a result, UDD enables people with less Debian expertise to mine Debian, lowering the entry barrier dramatically. While it was not the main goal for UDD, it is an interesting offspring.

B. Data Flow

The architecture of UDD, whose data flow is depicted in Figure 1, is rather straightforward. For each *kind* of data source (e.g. a specific bug tracking system or package archive) there is a *gatherer* software component which implements an abstract interface and abstracts over any data source instance. By using the gatherer API one can, intuitively, obtain a set of `INSERT` queries that performs the injection of a specific data source into UDD. The code of gatherers is where the complexity and heterogeneity of accessing the various data sources is hidden, shielding UDD users from them. The main UDD *updater* periodically invokes gatherers. Periods vary from gatherer to gatherer, to cope with data sources updated with varying frequencies. The updater is also responsible for maintaining some auxiliary tables and can be triggered out of band if needed.

C. Design Choices

The above described architecture descends from a few key design choices which are detailed below.

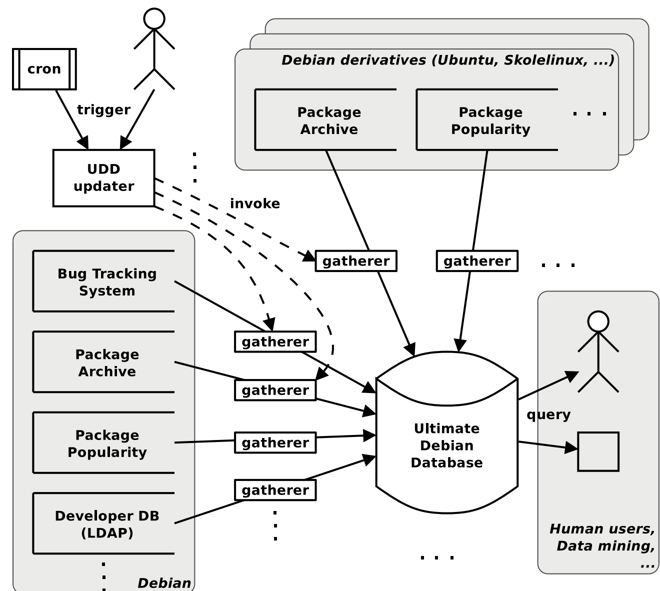

Figure 1. Data flow in the Ultimate Debian Database

General-purpose, user-friendliness: First, UDD has not been developed to answer a specific set of queries. While QA needs has provided an initial query test bench, the database schema is meant to be general, with as few assumptions as possible on the final user case. Second, when the choice was possible, staying as close as possible to the structure of the original data was favored to performance. For instance, surrogate keys[5] have been avoided and real data (package names, bug IDs, . . .) have been preferred, even when that has meant using an aggregation of columns as primary key. For users, which are usually familiar with the original data source, this simplifies query development. As evidence of that, we have witnessed several external contributions of new gatherers by people that were not involved in the initial development of UDD. According to contributors, the ability to query data via SQL and to join them with other data sources already injected into UDD was valuable.

Nevertheless, this choice does not hinder neither the development of applications using UDD as a basis, nor the optimization for those applications by the means of indexes.

Inconsistency preservation: Due to the nature of data organization in Debian (a lot of distributed services, without much shared data), inconsistencies can exist among services. For example, it is possible to report bugs against, and package popularity of nonexistent packages. Even more so, different services use different names for basic concepts such as *package*: while for archive management services, a package is defined as the 4-tuple

[5] A *surrogate key* is an arbitrary unique identifier (usually a sequential number) used as a key, which is not derived from any application data.

$\langle distribution, suite, package, version \rangle$, it is simply defined as the package name in the popularity contest.

In data integration, it is often tempting to remove inconsistencies from the data to simplify the modeling. In UDD inconsistencies have been preserved, because they are likely to be *interesting* for several use cases, including Quality Assurance. Inconsistencies provide a lot of useful information—e.g. one could investigate the packages that are installed on user systems, but are not provided by Debian, to find popular software that need official packaging—, but also harden writing correct queries. To mitigate this problem, *sanitized* versions of specific data are provided via `SQL VIEW`s as needed.

An important consequence of maintaining inconsistencies is that it is often not possible to link data from different data sources together with foreign keys. It might then be difficult for the user to determine how to use such data together. To mitigate this, UDD enforces consistent *naming conventions* for the same concept in spite of its origin data source. For example, an important distinction in Debian exists between source packages, which are called `source` in UDD, and binary packages, called `package`. This sometimes means drifting away from the original naming, e.g. in the `wanna-build` service (source package build management), where *source* packages are called `packages`.

Faithful gathering: Some form of correctness in data gathering is essential. For instance, various QA monitors are used to mail people about the status of their packages. Doing that on the basis of incorrect data can be perceived as SPAM-ing by recipient developers, possibly diminishing the usefulness of the monitor, and the likelihood of its acceptance in the community. Researchers data mining UDD have similar concerns. Since UDD cannot "fix" incorrectness in the external data sources, and given that time lags are the norm in data warehousing solutions, the notion of correctness for UDD is *faithful gathering*: after each gatherer invocation, tables are requested to contain a representation of the external data source at the time of gathering.

The main consequence of this desiderata is the avoidance of *partial updates*—i.e. processes in which deltas between the previous table status and the data source are computed, and then applied to obtain the new table status. The chosen alternative solution is to perform complete data reloads at each run. The additional benefit is in simplicity: UDD can always be thrown away and rebuilt from scratch by simply re-running all data gatherers. Since data reloads are potentially slower than incremental updates, transactions are used to always offer consistent snapshots to users.

Storage of historical data: Ideally, UDD aims at storing both current state and historical information about gathered data sources, given that *both* might be useful to pursue its goals. Unfortunately, due to size issues (see Section IV for some figures), we cannot currently afford to take periodic yet naive snapshots of all its tables, in a way that would enable querying historical data via SQL. Two partial solutions have been implemented to alleviate the drawbacks of such limitation. The first one is periodic publishing of complete database dumps, which are currently took every 2 days and made available via the UDD home page. Independent parties can periodically retrieve the dumps and inject them in external data warehousing solution with the resources to support the time axis. Interest by the FLOSSmole project [14] in doing that has been expressed already[6] and is being implemented.

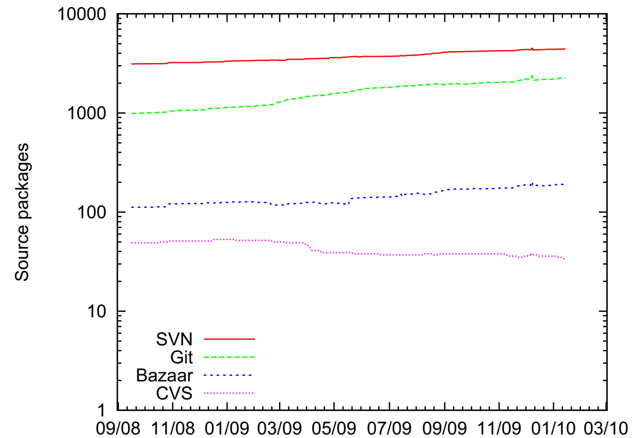

Figure 2. Historical evolution of Version Control System usage to maintain Debian source packages, based on UDD data.

Additionally, the choice of complete data reloads have been traded-off in a companion yet separate database called *UDD history*. Daily, the UDD updater takes snapshots of specific aggregated values and stores them in history tables where each tuple is associated to a validity timestamp. For instance, the `history.sources_count` table contains periodic snapshots of total source packages counts, such as how many packages are declaring a specific version control system, have been switched to new source formats, or even more simply the daily total number of packages in the archive. Interesting trends on package maintenance are being observed within Debian on the basis of such data, for instance Figure 2 shows the evolution of version control system usage for package maintenance in the last 2 years.

Unlike UDD, the history database cannot be recreated from scratch, therefore it is subject to specific backup care.

IV. Current Status

Currently, UDD sports 17 gatherers (see Table II), covering all the core parts of the Debian infrastructure, and several other parts that have been added on a by-need basis. The gatherers import data into a PostgreSQL database, which comprises 60 tables and over 7 million tuples, for a total size of 3.8 GB (excluding indexes). A full SQL dump is

[6] http://lists.debian.org/debian-qa/2009/06/msg00005.html

Table II
ULTIMATE DEBIAN DATABASE STATUS: AVAILABLE GATHERERS

Gatherer Name	Description and Notes	Corresponding UDD Tables
Sources and Packages	Information about source and binary packages for each Debian release and architecture, obtained parsing the RFC 2822-formatted `Sources` and `Packages` files. Also used to import the same information for Ubuntu to a different set of tables. Due to the size of the `packages` table, a `packages_summary` table is also provided, but contains less information (only the most useful fields, and no architecture-specific information).	`sources`, `packages`, `packages_summary`, `uploaders`, `ubuntu_sources`, `ubuntu_packages`, `ubuntu_packages_summary`, `ubuntu_uploaders`
Debian bugs	Status of all Debian bugs (archived or not), and their computed status (whether they affect specific Debian releases or not). The gatherer is written in Perl, leveraging the native API of the Debbugs bug tracking system.	`bugs`, `bugs_fixed_in`, `bugs_found_in`, `bugs_merged_with`, `bugs_packages`, `bugs_tags`, `bugs_usertags` and their `archived_bugs_*` counterparts, as well as `all_bugs_*` `VIEW`s
Ubuntu bugs	Status of all open bugs in Ubuntu. Imported using the Launchpad text interface, which still requires one HTTP request per bug (i.e. hammering).	`ubuntu_bugs`, `ubuntu_bugs_duplicates`, `ubuntu_bugs_subscribers`, `ubuntu_bugs_tags`, `ubuntu_bugs_tasks`
Popularity contest	Information about packages popularity (number of reported installations on user systems). Also used to import the same information for Ubuntu.	`popcon`, `ubuntu_popcon`
Upload history	History of package uploads (since 2002) to Debian unstable and experimental. Generated by parsing the mailing list archives for the `debian-devel-changes@` mailing list.	`upload_history`, `upload_history_architecture`, `upload_history_closes`
Upstream status (DEHS)	Latest version of the upstream software. Used to check that the Debian version is up-to-date.	`dehs`
Packages tags (`debtags`)	Set of per-package tags, used to classify packages in a faceted way.	`debtags`
Lintian results	Results of Lintian checks (a policy conformance checker)	`lintian`
Buildd status (`wanna-build`)	Status of each package on the build daemons (has it been built successfully? failed to build? waiting in the queue?)	`wannabuild`
Developers identities (`carnivore`)	Information about the different names, emails and PGP key each Debian developer or contributor uses, enabling to link different emails to the same developer.	`carnivore_emails`, `carnivore_keys`, `carnivore_login`, `carnivore_names`
Official developers (Debian LDAP)	List of official Debian developers, and information about their activity (last email, last upload, . . .)	`ldap`
Orphaned packages	List of packages that have been orphaned by their maintainer, with the time of the orphanage. Built from the `bugs` table and parsing of the bug logs.	`orphaned_packages`
Package removals	Information about packages that were removed from the Debian archive.	`package_removal`, `package_removal_batch`
Screenshots	Links to screenshots provided by screenshots.debian.net.	`screenshots`
Localized packages descriptions	Translated packages descriptions, provided by the ddtp.debian.net project.	`ddtp`
New packages	Packages currently being reviewed, before their acceptance in the Debian archive	`new_packages`, `new_sources`
Testing migrations	Historical information about packages migrations to the `testing` development suite	`migrations`

about 450 MB of `gzip`-ed SQL statements. Unsurprisingly, the `packages` table is the largest one (764,000 tuples, for about 1.4 GB), as it stores binary packages information for each recent Debian release and architecture (Debian has supported 10 architectures or more since 2002).

The UDD schema went through several iterations and improvements, and now matches our goals both regarding completeness (most of Debian data sources are imported into UDD) and ease of use. The full database schema is not reported in this paper due to space constraints, but is available on the UDD home page. The part of the schema about Debian bugs is shown in Figure 3 to give an idea of the challenges to be faced when implementing a gatherer.

A. Importing Debian bugs data

The Debian bug tracking system—called *Debbugs*—was initially developed more than 10 years ago. It is often seen as a dinosaur, because its web interface is limited to viewing the current status of a bug: all modifications to bug reports, including followups, happen via a mail interface. Debbugs data organization has not changed substantially since then. Each bug is internally stored using essentially two files:

- a *log file*, that contains the various emails that were exchanged about that bug, and

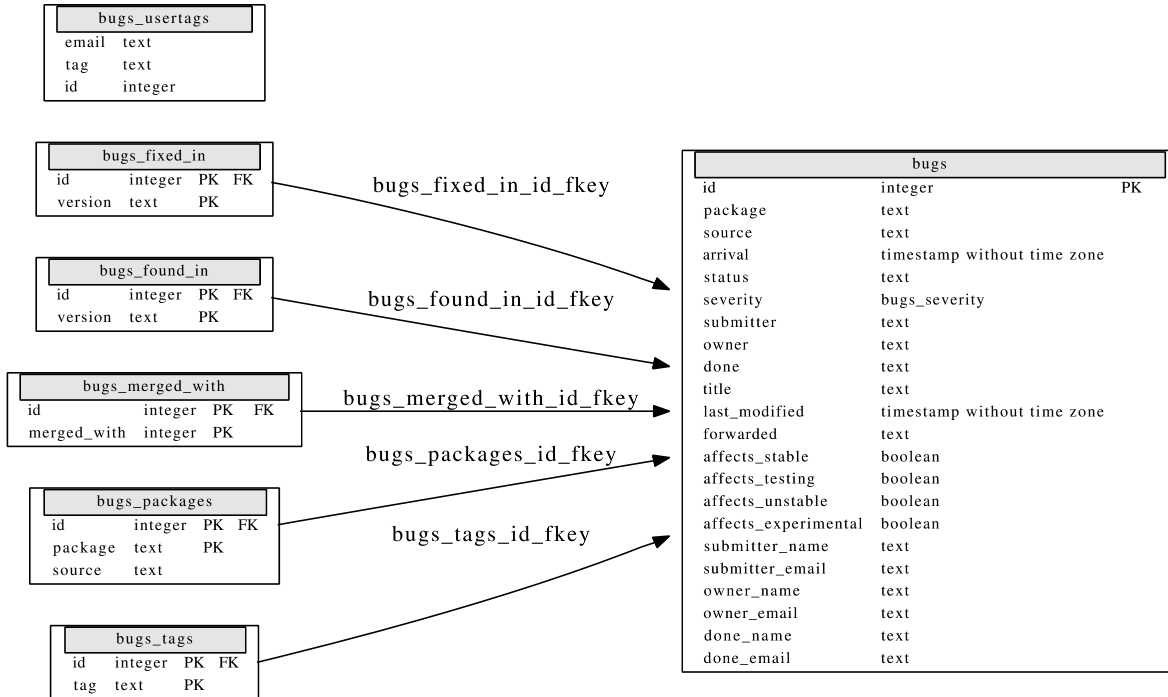

Figure 3. UDD schema corresponding to the Debian bug tracking system. All the information stored in the original service are available from UDD, as well as some convenience columns (source, affects_unstable, affects_testing, etc.) which are computed during data injection.

- a short *summary* file, that contains a few lines of metadata about the bug.

In UDD, the choice was made to only import bug metadata, ignoring bug logs. That requires reading 500,000+ summary files, which provides an interesting system programming challenge when efficiency is sought. "Obviously", the format of the summary file is only documented in Debbugs sources.

Unlike most BTSs where a bug has two distinguished open/closed states, in Debbugs the state of a bug depends on a specific version. It is hence possible to describe that a bug is fixed in the *unstable* suite, while the *stable* suite is still affected. Based on an analysis of the version tree for each package, and on the knowledge of the status of each suite, Debbugs computes which versions and suites are affected by the bug. Importing only the raw bug data into UDD, leaving the "affected by" computation to the user, would not be particularly user-friendly. Therefore, it was decided to do the computation for each bug during the import. As a result, the Debbugs gatherer imports raw information about versions marked as affected or unaffected (tables `bugs_fixed_in`, `bugs_found_in`), computes the status of each bug, and stores it (`bugs` table, columns `affects_stable`, `affects_testing`, etc.).

V. UDD Applications

In this section we give some representative examples of the possibilities offered by UDD.

A. Popular yet buggy packages

Let's start with the QA need of monitoring very popular and yet buggy packages that we have detailed in Example 1. That need is easily fulfilled by the following SQL query, which joins together tables about bugs, package popularity, and source packages:[7]

```
select sources.source, id, insts, title
from bugs
join sources on sources.source = bugs.source
join sources_popcon
  on sources_popcon.source = bugs.source
where severity >= 'serious'
  and distribution = 'debian'
  and release = 'squeeze'
  and affects_testing
order by insts desc
```

Excerpt of the query results are shown in Table III; the query execution time is $< 1s$.

B. Bapase: tracking neglected and orphaned packages

With more than 15,000 source packages, finding a needle in a haystack is not necessarily harder than finding those packages that have been neglected by their maintainers, or those that have been affected by important bugs for long. Being able to combine all the data sources that are imported

[7] `sources_popcon` is a convenience view exposing popcon for each source package, and using **max**`(insts)` to aggregate the results.

Table III
RESULTS OF THE QUERY IMPLEMENTING EXAMPLE 1 (EXCERPT OF).

source	id	inst	title
ncurses	553'239	87'420	missing-dependency-on-libc ...
ncurses	563'272	87'420	libtic.so dangling symlink ...
pam	539'163	87'415	profiles with no auth ...
		...	
fossology	550'653	2	depends on xpdf-utils
python-carrot	560'583	2	FTBFS: ImportError: ...

into UDD has provided a huge help to that end. *Bapase* (for *BAd PAckage SEarch*)[8] is a web interface to UDD that provides an overview of packages matching various QA criteria. The following criteria are currently implemented:

- *Orphaned packages*. When maintainers are no longer interested in a package, they can orphan it and put it up for adoption by another developer. However, over time, the number of orphaned packages tends to increase (because many of the orphaned packages were orphaned for a good reason, and are only of limited interest). By providing a global view of orphaned packages, together with information on popularity, age of orphanage, and number of bugs affecting the package, Bapase gives QA team members enough information to take a decision on the future of the package.
- *Packages maintained via NMUs*. When a maintainer is not responding in a timely manner to problems, it is possible for another developer to prepare and upload a fixed package. Such procedure is called *Non-Maintainer Uploads*. Packages which have been NMUed several times are likely to be neglected by their maintainers, and it might be a good idea to give the maintenance of such packages to more available developers.
- *Packages which fail to migrate to testing*. The Debian release process includes the migration of packages from the *unstable* to the *testing* suite, subject to quality requirements (e.g. a package must not have RC bugs to be eligible). When packages fail to migrate for a long time, it is often a sign of severe problems with the packages, or of negligence by the maintainer. Such packages are often good candidates for either removal from the Debian archive, or for maintainer change.
- *Packages not maintained by official developers*. The *sponsorship* process gives the possibility to non-developer contributors to maintain packages, by using an official developer as a proxy. While packages maintained by unofficial developers are not necessarily of low quality, it is not uncommon that such maintainers quickly lose interest in the packages they maintain. To spot the involved packages, a surprisingly large number of data sources are required. One need to get the list of official developers from the `ldap` table and combine it with the information from `carnivore` to get all the email addresses that official developers use. Then, using the `sources` and `uploaders` tables, packages maintained or co-maintained by official developers need to be excluded. Finally, the data common to all Bapase search need to be retrieved, using the `orphaned_packages`, `migrations`, `upload_history`, `bugs` and `popcon` tables. Despite the number of `JOIN`s involved, the query takes less than 20 seconds to execute.

In addition to the above monitoring, Bapase supports the process tracking the actions that should be taken by the QA team to improve the situation. Those two steps provide an unique tool to improve the quality of Debian by shedding some light on the darker areas of the package archive.

C. Collaboration with Debian derivatives

As we highlighted already, the Debian project is more than its distribution, it is rather the center of an ecosystem of derivative distributions which bring mutual benefits to all involved parties. As a data warehousing solution that does not incur additional costs in the injected data sources, UDD looks like the perfect place where to store for comparative processing information about all derivatives. This aspect is particularly interesting for Ubuntu, which is nowadays the major Debian derivative. This importance is already acknowledged by the current set of gatherers, which inject into UDD information about Ubuntu source and binary packages, Launchpad bugs, and popularity information. Having such information into UDD has enabled Debian developers with scarce information on Ubuntu inner workings to integrate Ubuntu versions and available patches in the two most popular developer dashboards: the Package Tracking System, and the Debian Developer's Packages Overview (DDPO).[9]

Example 2: As an example of a derivative need that can be easily fulfilled by UDD, the following query determines which packages are more recent in Debian than in Ubuntu, and which therefore need either synchronization or merging on the Ubuntu side:

```
select ubuntu.source, ubuntu.version,
  debian.version, insts
from ubuntu_sources ubuntu,
  sources_uniq debian,
  ubuntu_sources_popcon upopcon
where ubuntu.distribution = 'ubuntu'
  and ubuntu.release = 'lucid'
  and debian.distribution = 'debian'
  and debian.release = 'squeeze'
  and ubuntu.source = debian.source
  and ubuntu.version < debian.version
  and ubuntu.source = upopcon.source
order by insts desc
```

[8] http://udd.debian.org/cgi-bin/bapase.cgi

[9] http://qa.debian.org/developer.php

The query takes about $1.5s$ to execute. An enhanced version of it is used as the official list of merges in Ubuntu.[10]

Example 3: Debian benefits not only from code contribution by derivatives, but also from their precious feedback. To that end, the following query gives an overview of packages available in Ubuntu but not in Debian, sorted by their (Ubuntu) popularity. All such packages are good targets to be packaged in Debian too, most likely by simply reusing the already available packaging.

```
select src.source, insts
from ubuntu_sources src,
  ubuntu_sources_popcon pc
where distribution = 'ubuntu'
  and release = 'lucid'
  and src.source = pc.source
  and src.source not in
    (select source from sources
     where distribution = 'debian')
order by insts desc
```

The query runs in $< 1s$. Its live results are available on the UDD home page, together with those of others sample queries.[11]

VI. Related work

Data integration issues have been known for quite some time in the database research community [8]. In that respect, UDD is a data warehousing solution [15] for the Debian project. To the best of authors knowledge, it is the first solution discussing and taking into account the specifics of purely community-driven FLOSS distributions.

Several data mining studies have considered the Debian distribution as their subject due to its peculiarities in size, complexity, and history (e.g. [7], [10], [11], [12], [16], [17], [18]). The relationship of those works with the present one is manifold. We postulate that UDD will: (1) ease similar researches by enabling researchers to direct their efforts on data mining rather than acquisition, (2) enable more complex data mining on patterns emerging from *joint* data sources (as hinted by the 2010 MSR challenge [19] which proposed UDD as data source), (3) provide data which are to some extent *validated* by the Debian community, thus relieving researchers from risks like that of using outdated sources.

Other projects have been providing integrated data about FLOSS environments; most notably: FLOSSmole [14] (a collaborative repository of research data about FLOSS projects), SRDA [20] (periodic publishing of SourceForge.net project data), FLOSSMetrics [21] (a large scale database of information and metrics about FLOSS development), and an experience by Mockus to index source code data coming from of as many forges as possible [22]. A first difference among those efforts and UDD is in scope: UDD integrates data from distributions—and in particular from Debian-based distributions—rather than data from development projects. This has both disadvantages (e.g. RPM-based distributions are not tracked) and advantages, such as a more complete data span and freshness within the intended scope. The second relevant difference is in the ability and ease of joining data sources together, which is a key design principle of UDD. We consider the interest of FLOSSmole in UDD to be evidence of such advantages.

On the technical side, the only known data integration solutions that strictly relate to UDD are Launchpad and Mole. The former[12] is the all-in-one infrastructure developed by Canonical Ltd. for the Ubuntu distribution. Launchpad does offer an API, but one that is geared toward per-package actions, offering no facility to correlate data sources with an archive-wide scope. As evidence of that, the monitoring of which packages in Ubuntu need merging with Debian is currently based on UDD, which at present seems to be the best choice to data mine the Ubuntu distribution. To a less extent of integration, other company-backed distributions have infrastructures similar to Launchpad, with similar massive retrieval issues. Mole[13] is an effort to collect Debian data sources in a central place; as such it is very similar to UDD and in fact has inspired it. However, Mole has made the design choice to offer as its API a set of BerkeleyDB files published on the web. That choice hinders access (data must be downloaded locally), data correlation (joins have to be made by hand), and fire-and-forget testing (SQL-like interactive query environments are scarce). As a result, UDD has nowadays more data source than Mole, fresher data, and more users.

VII. Conclusions

The Ultimate Debian Database (UDD) is a data warehousing solution for data sources about the Debian project. It was born to solve the so called "data hell" problem—an intrinsic difficulty in correlating distribution data to highlight package and maintainer metrics which are relevant for Quality Assurance (QA)—, but it is by no means QA-specific. UDD is as suited for data mining research as it is for QA, which is in fact purpose-specific data mining. The generality of UDD design choices, together with the research attention on Debian which is not fading after 15 years of history, offers an unparalleled easy to use platform to establish facts about Debian. From an ethical point of view, UDD makes Debian be a better citizen in both the FLOSS and research ecosystems, by providing an open interface to all of its data—a feature that company-based distributions are usually unwilling to offer. That aspect can be leveraged by derivative distributions to import Debian data in their own infrastructures.

The main current limitation of UDD is scarce resource availability, which makes impractical to store the history of

[10]http://people.ubuntuwire.com/~lucas/merges.html

[11]http://udd.debian.org/cgi-bin/

[12]https://launchpad.net

[13]http://wiki.debian.org/Mole

all of its data. A companion database and future cooperation with FLOSSmole contribute to diminish the resulting disadvantages. Another limitation descends from the sampling nature of data injection, which is common to data warehousing solutions: data are periodically pulled *by* UDD, rather than pushed *to* it as soon as they change. There is therefore no guarantee that all data will be eventually available in UDD. More tightly integrated data schemes, where data are granted to be fed into UDD at each change, do not seem feasible at present according to the peculiarities of the Debian infrastructure.

We look forward for new exciting data mining discoveries about the Debian project, powered by UDD!

Acknowledgments

The authors would like to thank all UDD contributors, and in particular: Christian von Essen and Marc Brockschmidt (student and co-mentor in the Google Summer of Code which witnessed the first UDD implementation), Olivier Berger for his support and FLOSSmole contacts, Andreas Tille who contributed several gatherers, the Debian community at large, the "German cabal" and Debian System Administrators for their UDD hosting and support.

References

[1] A. Deshpande and D. Riehle, "The total growth of open source," in *OSS 2008: 4th Conference on Open Source Systems*. Springer, 2008, p. 197.

[2] H. Kagdi, M. Collard, and J. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, pp. 77–131, 2007.

[3] J. Howison and K. Crowston, "The perils and pitfalls of mining SourceForge," in *MSR 2004: International Workshop on Mining Software Repositories*, 2004, pp. 7–11.

[4] R. Di Cosmo, P. Trezentos, and S. Zacchiroli, "Package upgrades in FOSS distributions: Details and challenges," in *HotSWup'08*, 2008.

[5] L. Nussbaum, "Rebuilding Debian using distributed computing," in *CLADE'09: 7th International Workshop on Challenges of large applications in distributed environments*. ACM, 2009, pp. 11–16.

[6] M. Michlmayr and B. M. Hill, "Quality and the reliance on individuals in free software projects," in *3rd Workshop on Open Source Software Engineering*, 2003, pp. 105–109.

[7] J. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. Amor, and D. German, "Macro-level software evolution: a case study of a large software compilation," *Empirical Software Engineering*, vol. 14, no. 3, pp. 262–285, 2009.

[8] M. Lenzerini, "Data integration: a theoretical perspective," in *PODS'02: 21st Symposium on Principles of Database Systems*. ACM, 2002, pp. 233–246.

[9] E. Raymond, *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary*. O'Reilly, 2001.

[10] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor, "Mining large software compilations over time: another perspective of software evolution," in *MSR 2006: 3rd International Workshop on Mining Software Repositories*. ACM, 2006, pp. 3–9.

[11] I. Herraiz, G. Robles, R. Capilla, and J. M. González-Barahona, "Managing libre software distributions under a product line approach," in *COMPSAC*, 2008, pp. 1221–1225.

[12] P. Abate, R. D. Cosmo, J. Boender, and S. Zacchiroli, "Strong dependencies between software components," in *ESEM 2009: 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 89–99.

[13] M. Michlmayr, "Managing volunteer activity in free software projects," in *2004 USENIX Annual Technical Conference, FREENIX Track*, 2004, pp. 93–102.

[14] J. Howison, M. Conklin, and K. Crowston, "FLOSSmole: A collaborative repository for FLOSS research data and analyses," *International Journal of Information Technology and Web Engineering*, vol. 1, no. 3, pp. 17–26, 2006.

[15] D. Theodoratos and T. Sellis, "Data warehouse configuration," in *VLDB 1997: 23rd International Conference on Very Large Data Bases*. ACM, 1997, pp. 126–135.

[16] J. González-Barahona, M. Perez, P. de las Heras Quirós, J. González, and V. Olivera, "Counting potatoes: the size of Debian 2.2," *Upgrade Magazine*, vol. 2, no. 6, pp. 60–66, 2001.

[17] T. Maillart, D. Sornette, S. Spaeth, and G. Von Krogh, "Empirical tests of Zipf's law mechanism in open source Linux distribution," *Physical Review Letters*, vol. 101, no. 21, p. 218701, 2008.

[18] N. LaBelle and E. Wallingford, "Inter-package dependency networks in open-source software," *Submitted to Journal of Theoretical Computer Science*, 2005.

[19] A. Hindle, "MSR mining challenge," http://msr.uwaterloo.ca/msr2010/challenge/, 2010, retrieved 12/01/2010.

[20] "SRDA: SourceForge.net research data," http://www.nd.edu/~oss/Data/data.html, retrieved 12/01/2010.

[21] I. Herraiz, D. Izquierdo-Cortazar, and F. Rivas-Hernández, "FLOSSMetrics: Free/libre/open source software metrics," in *CSMR*. IEEE, 2009, pp. 281–284.

[22] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *MSR 2009: 6th International Workshop on Mining Software Repositories*, 2009, pp. 11–20.