Collection of Activity Data for SourceForge Projects

Scott Christley and Greg Madey

Technical Report: TR-2005-15
Dept. of Computer Science and Engineering
University of Notre Dame

October 10, 2005

1 Introduction

Social network analysis [Scott, 2000] [Wasserman and Faust, 1994] applied to understanding the Open Source Software development community requires that a sufficiently rich social network be constructed from the available data. While the social network derived from basic membership data, i.e. association of users to projects, allows global properties of the community to be calculated [Xu et al., 2005a] [Xu et al., 2005b]; such a social network is not sufficient to gain a detailed understanding of the development processes occurring within and across projects. For this reason, we have embarked on collecting a complete, detailed record of activities performed by users for open source projects. Specifically, we use the data dump of the "back-tier" relational database that drives the SourceForge.net website, and we collect all of the various activity records within this database and combine them together into a single database table. This report acts as a record of data provenance for [Christley and Madey, 2005], and portions of this data was used as part of the data set in [Tynan et al., 2005].

SourceForge.net[Sourceforge, 2005] is the world's largest Open Source software development web site, with the largest repository of Open Source code and applications available on the Internet. Owned and operated by OSTG, Inc. ("OSTG"), SourceForge.net provides free services to Open Source developers. The SourceForge.net web site is database driven and the supporting database includes historic and status statistics on over 100,000 projects and over 1 million registered users' activities at the project management web site. OSTG has shared certain SourceForge.net data with the University of Notre Dame for the sole purpose of supporting academic and scholarly research on the Free/Open Source Software phenomenon. There are over 100 relations (tables) in the data dump provided to Notre Dame. Some of the data have been removed for security and privacy reasons. SourceForge.net cleanses the data of personal information and strips out all OSTG specific and site functionality specific information. On a monthly basis, a complete dump of the database (minus the data dropped for privacy and security reasons) is shared with Notre Dame. The Notre Dame researchers have built a data warehouse comprised of these monthly dumps, with each stored in a separate schema. Thus, each monthly dump is a shapshot of the status of all the SourceForge.net

projects at that point in time. As of June 2005, the data warehouse was over 200 GBytes in size, and is growing at about 30 GBytes per month. Much of the data is duplicated among the monthly dumps, but trends or changes in project activity and structure can be discovered by comparing data from the monthly dumps.

2 Collection of Activity Data

Data for most of the activities performed by developers and users for projects is scattered across the many tables in the SourceForge database schema. In order to efficiently analyze this data, it is useful to collect all of the records together from the many source tables and put them into a small set of tables; we have written a database program which collects this information together. The schema for the activity tables is shown in Figure 1. The data for CVS activity is stored in the *user_project_cvs_act* table separate from the data for other activity which is stored in the *user_project_act* table. CVS activity is acquired from the CVS repositories unlike the other activity which is acquired from the SourceForge database; the CVS activity is also considerably larger, so queries can be directed against one or the other table depending upon the type of data desired. The *activity_name* table is a list of all the activity types that have been collected; these activity types are summarized in the following sections and in Table 1. As of the data of this article, the activity data collected includes all activity up to the June 2005 database snapshot of SourceForge.net.

2.1 Anonymous Users

Some projects allow anonymous users to perform activities related to the project; these may be activities like creating artifact records or posting forum messages. In the collection of activity data, we maintain the activity records performed by anonymous users; therefore in the analysis of this data, one has to be aware of whether anonymous activity should be used or not. Anonymous activity has the primary issue of lacking identity; it may be that a project member or registered user performed the activity but forgot to login to SourceForge, and it is unknown if multiple activities are performed by one, a few, or many people. Anonymous users are given user id 100 in the activity records, so that activity can be easily excluded if desired.

2.2 Artifact Data

Artifact is a generic term to represent numerous types of reports that are attached to a project. SourceForge automatically defines a number of default artifacts like bug reports, feature requests, support requests, and patches; however, projects can also define their own type of artifacts. Figure 2 show the set of tables in the SourceForge schema related to artifact records.

We want to separate out the different artifact types into separate activity records, so that we can analyze specific types of artifacts like just bug reports. To do this, we perform a two part processing of the artifact data. First we collect together all the desirable data by joining together the various

artifact tables and put that data into the *artifact_all* table; the query which joins the artifact data together is show in Figure 8.

Next we process all of the records in the *artifact_all* table by selecting out artifact types with string operations, and we insert the records into the activity table under the appropriate activity type. Figure 9 shows the set of queries used to extract bug reports, support requests, patches, feature requests, TODO lists, and other user-defined artifact types. The first four artifact types are default types defined by SourceForge, so they adhere to a common name; however, the TODO list is not a default type but a very common user-defined artifact type which is extracted with string operations to handle differences in case and punctuation. The remaining artifact types cannot be easily distinguished into significant, separate types so this activity is lumped together under the other artifact type.

Each artifact record actually has two users associated with it; one user is the person who created or submitted the artifact record, and the other user is the person assigned to the artifact record. Creation of an artifact record is initiated by the user; however, assignment of an artifact record to a user may not be initiated by that user. There are two primary ways for an artifact record to be assigned to a user; one is automatic assignment whereby the SourceForge project is setup such that new artifact records of a specific type are automatically assigned to a particular user. The other way is for a third party to assign the artifact record; in software engineering practice this is termed triage, incoming artifact records are reviewed by one (or many) people who then triages or assigns the artifact records to appropriate personnel. Triage is a division of labor technique that tries to insure that highly specialized technicians spend their time working on issues related to their specialty, not searching through artifact records for appropriate work. Therefore, it is important to be aware that the assigned artifact activity might not actually be performed by that user. Work is continuing to provide more detailed activity information regarding people actually performing the triage work. For each artifact record, we create two activity records, one for the submission of the artifact record and another for the assignment.

2.3 Forum Data

The message forums is a facility provided by SourceForge.net that allows users to post messages that other users can read; either new messages or followup messages to existing messages can be posted. A project can have multiple forums, possibly to separate discussions into different categories, and forums can be private allowing only project members to read and post messages or public allowing the general public to read and post messages. Figure 3 shows the tables in the SourceForge database related to the message forums.

When collecting forum activity, we differentiate between new messages and followup messages as two different activity types. The SQL query performed to collect the forum activity can be seen in Figure 10, and the *is_followup_to* field is used to determine if the message is a new message or a followup.

2.4 Project History Data

The project history table records activity information whenever a project member makes a change to the settings for a project. This activity includes adding/removing members, changing permissions, and changing of other various project settings. In our initial collection of activity data, we processed this information into activity records; unfortunately, the history table was dropped from the SourceForge schema somewhere between 2003 and 2005, so it is no longer possible to obtain this data. While we have the activity for the initial SourceForge data dump of 2003; none of the recent monthly data dumps have this information.

2.5 File Release Data

A file release is the activity of making a source code or binary code release of the software for a project. As open source projects on SourceForge, the source code for the project can generally be obtained at any time directly from the CVS repository; however, many projects consider the source code in CVS to be in the process of development, so such code may or may not function properly depending upon the state of development. To give users a stable version of the software, the source code is generally packaged up into a file release and made available. How often projects perform file releases, the stability of the software in a file release, or whether a file release is even made varies from project to project depending upon their internal development process. SourceForge.net provides the capability for projects to manage file releases as well as attach versioning, packaging, and status information to the file releases. The schema in the SourceForge database related to file release is shown in Figure 4. The query which collects the file release activity data is shown in Figure 11.

2.6 Project Task Data

SourceForge provides the capabilities for task management for projects. Project tasks can be categorized together into groups, can be broken up into subtasks which have dependencies upon other tasks, and have attributes like priority, status and start/end dates. The task management functionality is not a complete project management system as it lacks capabilities for resource and personnel management. The schema in the SourceForge database related to project tasks is shown in Figure 5.

We collect three types of activity records from project task tables: the creation of new project task, the modification of an existing project task, and the assignment of the project task. Unlike with artifact data, multiple people can be assigned to a single project task. The queries for collecting records for creating, modifying, and assignment of project tasks are shown in Figure 12, 13, and 14.

2.7 Documentation File Data

Many open source projects have the documentation for their project included with the source code, but this documentation often needs to be processed by text processing programs to put it in an end-user accessible and readable format. However, projects that maintain their own web site tend to provide those end-user accessible documents on their web site, but not all projects maintain a web site. Therefore, SourceForge provides the capability for projects to post simple documents in either ASCII text or HTML format onto the web. Documents can be categorized into documentation groups, and each has an associated natural language. Projects may also open up the document submission page to the public so that users can submit documentation to the project; the primary documentation activity is the creation of a new document, and there does not appear to be any mechanism for updating a document. The schema in the SourceForge database related to documentation files is shown in Figure 6, and the query which collects the documentation activity data is shown in Figure 15.

2.8 Project Jobs Data

Numerous open source projects are looking for new people to join the project and contribute in some fashion. SourceForge provides the facility for projects to post jobs; they are essentially help-wanted ads looking for people with particular skills and interests. Such project jobs are assigned to a general category like developer, tester, documentation writer, web designer, etc; and each job has a status indicating whether the position is open or has been filled. Requests for people with specific skills is specified with the skill inventory list; it indicates the type of skill like experience with a particular programming language, operating system, or technology, and it specifies the skill level and years of experience desired for that type of skill.

There is a single activity associated with project jobs which is the creation of the job entry; it is unclear whether it is possible to determine when a job has been filled along with an associated date and user. The query for collecting creation records is shown in Figure 16, and the schema in the SourceForge database related to project jobs is shown in Figure 7.

2.9 CVS Activity

Concurrent Versions System(CVS) activity is different from the other activities in that the data is not within the back-end database for the SourceForge website; instead, the data is within the history records in the CVS repository for each project. In order to collect this activity, we need to perform a CVS command to retrieve the history records, parse those history records to extract the user id, date, time, and the CVS operation performed, then insert appropriate activity records. CVS commands fall under two main categories; the first is considered typical developer activity and other is version or release management activity.

To understand the various CVS activities, it is worthwhile to provide some explanatory details about how CVS manages source code. The Concurrent Versions System is an open source software package

for managing a source code repository; while it is the most widely used software for open source projects, other software exists like the open source Subversions package and commercial applications like Clearcase and PVCS. Many of the commercial packages contain build management tools which means that besides managing source code, they also provide capabilities for compiling and building the source code into a final product. CVS only manages source code, and build management for open source projects often use the GNU autoconf and make packages. The primary reason for a source code repository is to allow multiple developers to be working on the source code at the same time without conflict and to manage the changes to the source code made by the developers.

When a developer desires to make source code changes, he must first check out the source code from the CVS repository. The command to check out the source code provides the developer with a complete local copy of all of the source code; this is often termed a sandbox, because the developer may make any modifications to his local copy and it will not affect other developers nor the source code in the repository. Possible modifications include adding new source code files, removing source code files, or changing the contents of existing source code files. By working in a sandbox, the developer can modify the source code and test those changes without conflicting with other developers; therefore, multiple developers can easily work concurrently on the software project because any changes are local to their sandbox. Once a developer has finished work and is ready to make the changes permanent, then the commit CVS command is used; it will take files from the developer's local sandbox and upload them to the source code repository thus making those changes available to other developers and users. If a developer has been working on the source code in his sandbox for a long time, many changes may be committed to the source code repository by other developers during that time; by default those committed changes are not automatically copied into the developer's sandbox. Instead the developer must explicitly ask CVS for those changes by using the update command; CVS will then update the local copy of the source code with any changes in the repository. Sometimes, changes committed by another developer conflict with changes that the developer has made in his local sandbox; during the update process, CVS will recognize any conflicts and attempt to resolve them automatically; this is generally called merging changes. If CVS is not able to automatically resolve the conflict, both the changes from the repository and the developer's local changes are marked and placed alongside each other, and the developer is instructed to manually merge the changes. A developer must manually perform that merge process before CVS will allow the developer to commit his changes to the repository.

One of the primary reasons why a source code repository like CVS is useful versus just saving the files to a common place like on a file server is that CVS maintains historical versions of all of the source code; an example illustrates how this may be used. Suppose a developer makes a change to a source code file then commits it to repository, and later it is discovered that the change was incorrect and needs to be removed; well without prior versions of the source code, there is no way to know exactly what those changes were except the memory of the developer (which may not be good). This example has two scenarios, in one case the error is caught quickly and it is just a matter of reverting the source code back to the previous version; in the more difficult case, numerous other changes are committed to the source code so reverting to a prior version will also lose the other changes, which is generally not desired. CVS handles both situations easily; because it maintains all version of the source code, it easy to revert to a previous version, and it can show the source code differences between two historical versions. So in the more difficult later case, the difference between two historical versions is retrieved, before and after the errant committed code, and only those changes are removed leaving any new changes in tact.

Besides providing capabilities for developers to concurrently work on source code, CVS also provides commands that would be considered useful for release management. The CVS export command is similar to the checkout command in that it provides a local copy of the source code except that various CVS administrative files present with checkout are not present with export, so the export command gives a pristine copy of the source code. This is often used to make a source code release of the software project available to users who cannot directly access the CVS repository. The CVS tag command performs a dual function of marking a specific version of the source code with a label and allowing for source code to be branched. Tagging the source code with a label allows for that specific version to be referenced by the label without remembering the internal CVS version numbers; for example, when a new version of the software is released, the source code may be tagged with a label for that version like **MY_PROJECT_VERSION_1**. Later if some user has an issue with that version of the software, developers are able to retrieve the exact source code, based upon the label name, even though many changes may have been committed to the repository in the mean time. Tagging also allows for branching which is a more sophisticated mechanism for supporting concurrent development of multiple version of the software. In the above case, after the release of version 1 of the software, the developers want to start working on version 2 while still allowing for bug fixes to version 1; this is accomplished by branching the source code from that label to create a version 1 branch. Now there are two parallel forks of the code which are independent of each other, the version 1 branch and the main (version 2) branch; developers can commit changes to each of the two branches separately without conflict. Bug fixes can be committed on the version 1 branch while new development occurs on the main branch. This example of using tags and branches is only possible process for managing multiple versions of a software projects; CVS supports any number of branches, so there are various other processes that projects can implement.

3 Application of Activity Data

3.1 Construction of a Social Network from Activity Data

In [Christley and Madey, 2005], we construct a multi-relational, weighted, bipartite (two-mode or affiliation) network from the non-CVS activity data described in this document. In this network the nodes are individuals and projects, and the edges link individuals to projects. Each edge between an individual and a project represents a relationship based upon an activity that individual performs for that project, and the weight on the edge corresponds to the quantity of that activity performed by the individual. Multiple edges can exist between an individual and a project where each edge represents a different relationship; we define the different relationships as different types of activities the individual can perform.

Networks can be represented in multiple ways within a computer program; the different representations have different tradeoffs for the amount of memory used to store the network and the speed which certain operations can be performed. The two primary representations are the adjacency matrix and the adjacency list. The adjacency matrix is a two-dimensional matrix with network nodes on both the rows and the columns, and each matrix cell holds a value that indicates whether an edge exists between the corresponding row and column. The adjacency list is a list of all nodes in the network where each node has a list that contains the edges connecting it to other nodes.

The adjacency matrix requires more space because each matrix cell is represented regardless of whether an edge exists or not; however, accessing edges in the network is very fast. The adjacency matrix is very space efficient because only the edges that exist are in the lists; however, accessing edges is slower because the list must be traversed. For very large matrices and also for sparse matrices, the adjacency list representation is preferred, and sometimes it is the only choice for large matrices because the adjacency matrix cannot fit in the available memory. The social network constructed from the activity data is a very large and sparse network, so we use the adjacency list representation.

The following Java code segment shows how we construct the social network from the activity data. Note that a number of queries are involved; some of them find counts so that the lists (arrays) can be allocated of the correct size; while, others retrieve the actual data to put into the lists. Another technique being used is a hash table to map from a user id number to an index in the list of all users; this allows for efficiently finding the user in the list without searching the list from the beginning. As this is a bipartite network, we keep a list of users separate from the projects; all of the edges going from users to projects, so each user has its own list of projects. In fact, each user actually keeps two lists; one which contains the project id numbers for each project, and another list which holds the weight (count) for the different activity types for each project.

```
System.out.println("Loading user-project activity network from database.");

// get total number of users
String query = "select count(distinct user_id) from user_project_act "
               + "where user_id != 0 and user_id != 100";
PreparedStatement stmt = load_conn.prepareStatement(query);
stmt.execute();
ResultSet rset = stmt.getResultSet();
rset.next();
int totUsers = rset.getInt(1);
stmt.close();

// get total number of activities
query = "select count(activity) from activity_name";
stmt = load_conn.prepareStatement(query);
stmt.execute();
rset = stmt.getResultSet();
rset.next();
int totActs = rset.getInt(1);
stmt.close();

// allocate user arrays
Hashtable userMap = new Hashtable();
int userTo[][] = new int[totUsers][];
int userActivity[][][] = new int[totUsers][][];

// get number of projects for each user
query = "select user_id, count(distinct project_id) "
```

```
                + "from user_project_act "
                + "where user_id != 0 and user_id != 100 "
                + "group by user_id";
stmt = load_conn.prepareStatement(query);
stmt.execute();
rset = stmt.getResultSet();

int entry = 0;
while (rset.next()) {
    int ui = rset.getInt(1);
    Integer userId = new Integer(ui);
    int projCnt = rset.getInt(2);

    // put user in map, allocate project and activity arrays
    userMap.put(userId, new Integer(entry));
    userTo[entry] = new int[projCnt];
    userActivity[entry] = new int[projCnt][];
    for (int i = 0;i < projCnt; ++i) {
        userActivity[entry][i] = new int[totActs];
    }
    ++entry;
}

// get counts of all activities for all users/projects
query = "select user_id, project_id, activity, count(activity) "
                + "from user_project_act "
                + "where user_id != 0 and user_id != 100 "
                + "group by user_id, project_id, activity "
                + "order by user_id, project_id";
stmt = load_conn.prepareStatement(query);
stme.execute();
rset = stmt.getResultSet();

int lastUser = 0;
int lastGroup = 0;
entry = 0;
int groupEntry = 0;
while (rset.next()) {
    int ui = rset.getInt(1);
    int gi = rset.getInt(2);
    int ai = rset.getInt(3) - 1;
    int actCnt = rset.getInt(4);

    Integer mapId = (Integer)userMap.get(new Integer(ui));
    int userId = mapId.intValue();

    // put project in array
    if (ui != lastUser) {
```

```
        entry = 0;
        lastUser = ui;
        lastGroup = gi;
        groupEntry = 0;
    } else
        ++entry;
    if (gi != lastGroup) {
        ++groupEntry;
        lastGroup = gi;
    }
    userTo[userId][groupEntry] = gi;

    // put activity count in array
    userActivity[userId][groupEntry][ai] = actCnt;
}
stmt.close();

System.out.println("Total users: " + totUsers);
```

After running the above code, the following set of variables contain the complete social network which can then be used for analysis. This data is for all of the activity across the lifetime of all users and projects. If you are interested in activity for just a certain period of time then you should alter the queries to limit the activity based upon a date range.

- totUsers is the total number off users.
- totActs is the total number of activity types.
- userMap is the Java Hashtable from user id to array index.
- userTo is the Java array with the list of project id numbers for each user.
- userActivity is the Java array with the weights (count) for each activity type for each project id for each user.

References

[Christley and Madey, 2005] Christley, S. and Madey, G. (2005). An algorithm for temporal analysis of social positions. In *North American Association for Computational Social and Organizational Science (NAACSOS 2005)*, Notre Dame, IN.

[Scott, 2000] Scott, J. (2000). *Social Network Analysis: a handbook*. SAGE Publications, 2nd edition.

[Sourceforge, 2005] Sourceforge (2005). http://www.sourceforge.net.

[Tynan et al., 2005] Tynan, R., Madey, G., Christley, S., Freeh, V., and Xu, J. (2005). Task characteristics, communication, and outcome in open source software development. *(unpublished)*.

[Wasserman and Faust, 1994] Wasserman, S. and Faust, K. (1994). *Social Network Analysis: Methods and Applications*. Cambridge University Press, Cambridge, UK.

[Xu et al., 2005a] Xu, J., Christley, S., Gao, Y., and Madey, G. (2005a). A topological analysis of the open source software development community. In *The 28th Annual Hawaii International Conference on System Sciences*, Hawaii.

[Xu et al., 2005b] Xu, J., Christley, S., and Madey, G. (2005b). The open source software community structure. In *North American Association for Computational Social and Organizational Science (NAACSOS 2005)*, Notre Dame, IN.

A Tables and Figures

The database schemas in the various figures in this section attempt to follow the standard conventions for Entity-Relationship diagrams. Each entity, which corresponds to a database table, is enclosed within a box; the name of the entity is given at the top of the box, and the lower section of the box lists the attributes for the entity. Each attribute has a name, a datatype, and code indicating if the attribute is a primary key (PK), a foreign key (FK), or not a key in which case the code is blank. A primary key means that attribute (or set of attributes) uniquely identifies each record in that database table. A foreign key means that attribute is a primary key for another entity, and each foreign key has a line connecting it to that foreign entity. Primary and foreign keys are used to join together entities when querying data. The arrows on the lines have significance to indicate if the join is a to-one or a to-many relationship; a to-one relationship is indicated by a single arrow, and a to-many relationship is indicated by a double arrow. A to-one relationship means that only a single record in the entity is returned as part of the join; while, a to-many relationship means that multiple records can be returned. For example, in Figure 1, the *activity* attribute in the *user_project_act* table has a to-one relationship with the *activity_name* table; therefore, a single record in the *user_project_act* only has one *activity* associated with it. However, the line has a double arrow on the *user_project_act* table indicating a to-many relationship, so a single record in the *activity_name* table can have multiple records in *user_project_act*. This relationship should correspond to our understanding of the data; any single activity record only has one type of activity, but there may be many activity records which all have the same type.

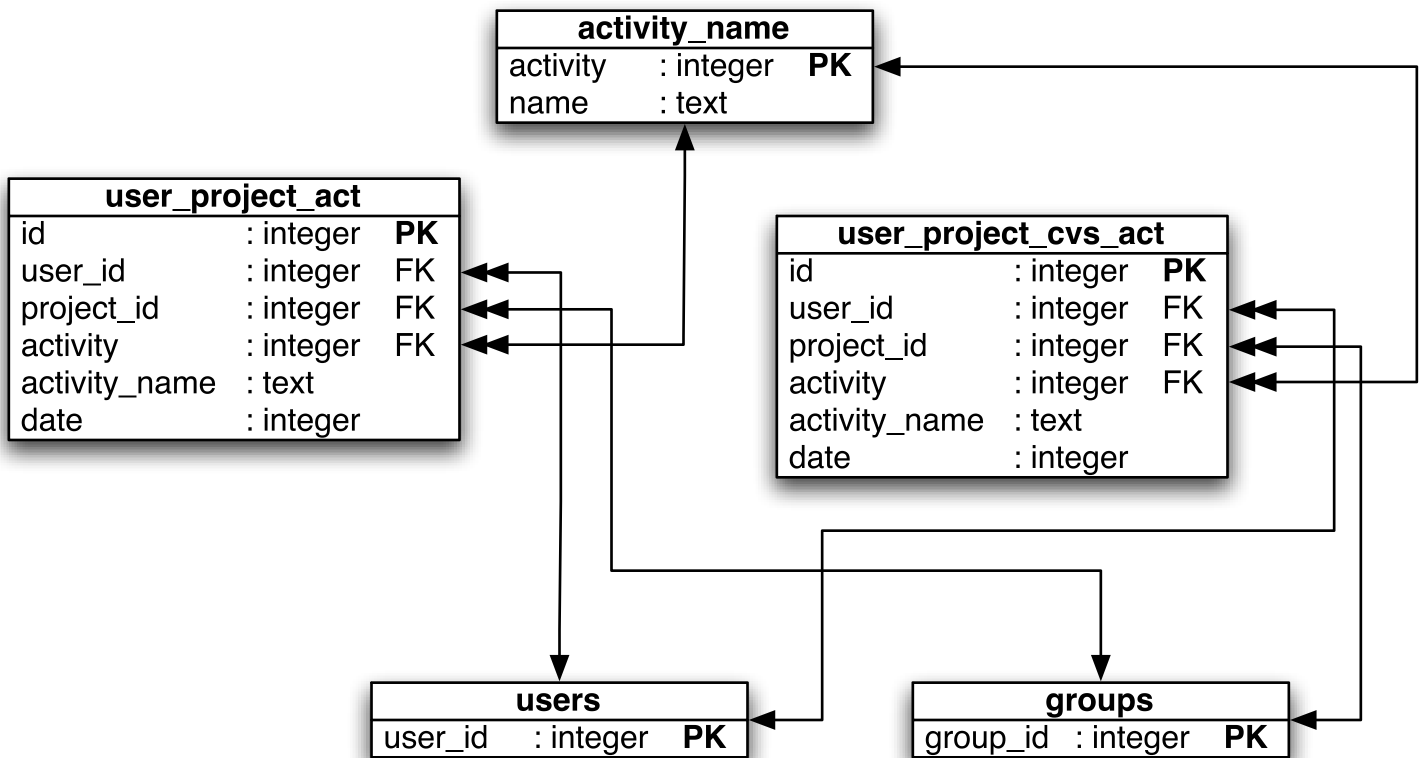

Figure 1: Schema for Activity Data

artifact		
artifact_id	: integer	PK
group_artifact_id	: integer	FK
status_id	: integer	FK
category_id	: integer	FK
artifact_group_id	: integer	FK
resolution_id	: integer	FK
priority	: integer	
submitted_by	: integer	FK
assigned_to	: integer	FK
open_date	: integer	
close_date	: integer	
summary	: text	
details	: text	
closed_by	: integer	FK

artifact_status		
id	: integer	PK
status_name	: text	

artifact_category		
id	: integer	PK
group_artifact_id	: integer	FK
category_name	: text	
auto_assign_to	: integer	FK

artifact_group		
id	: integer	PK
group_artifact_id	: integer	FK
group_name	: text	

artifact_resolution		
id	: integer	PK
resolution_name	: text	

artifact_file		
id	: integer	PK
artifact_id	: integer	FK
description	: text	
bin_date	: text	
filename	: text	
filesize	: integer	
filetype	: text	
adddate	: integer	
submitted_by	: integer	FK

artifact_group_list		
group_artifact_id	: integer	PK
group_id	: integer	FK
name	: text	
description	: text	
is_public	: integer	
allow_anon	: integer	
email_all_updates	: integer	
due_period	: integer	
use_resolution	: integer	
submit_instructions	: text	
browse_instruction	: text	
datatype	: integer	
status_timeout	: integer	
due_period_initial	: integer	
due_period_update	: integer	

artifact_history		
id	: integer	PK
artifact_id	: integer	FK
field_name	: text	
old_value	: text	
mod_by	: integer	FK
entrydate	: integer	

artifact_message		
id	: integer	PK
artifact_id	: integer	FK
submitted_by	: integer	FK
from_email	: text	
adddate	: integer	
body	: text	

artifact_perm		
id	: integer	PK
group_artifact_id	: integer	FK
user_id	: integer	FK
perm_level	: integer	

users		
user_id	: integer	PK

groups		
group_id	: integer	PK

Figure 2: Schema for Artifact Data in SourceForge Database

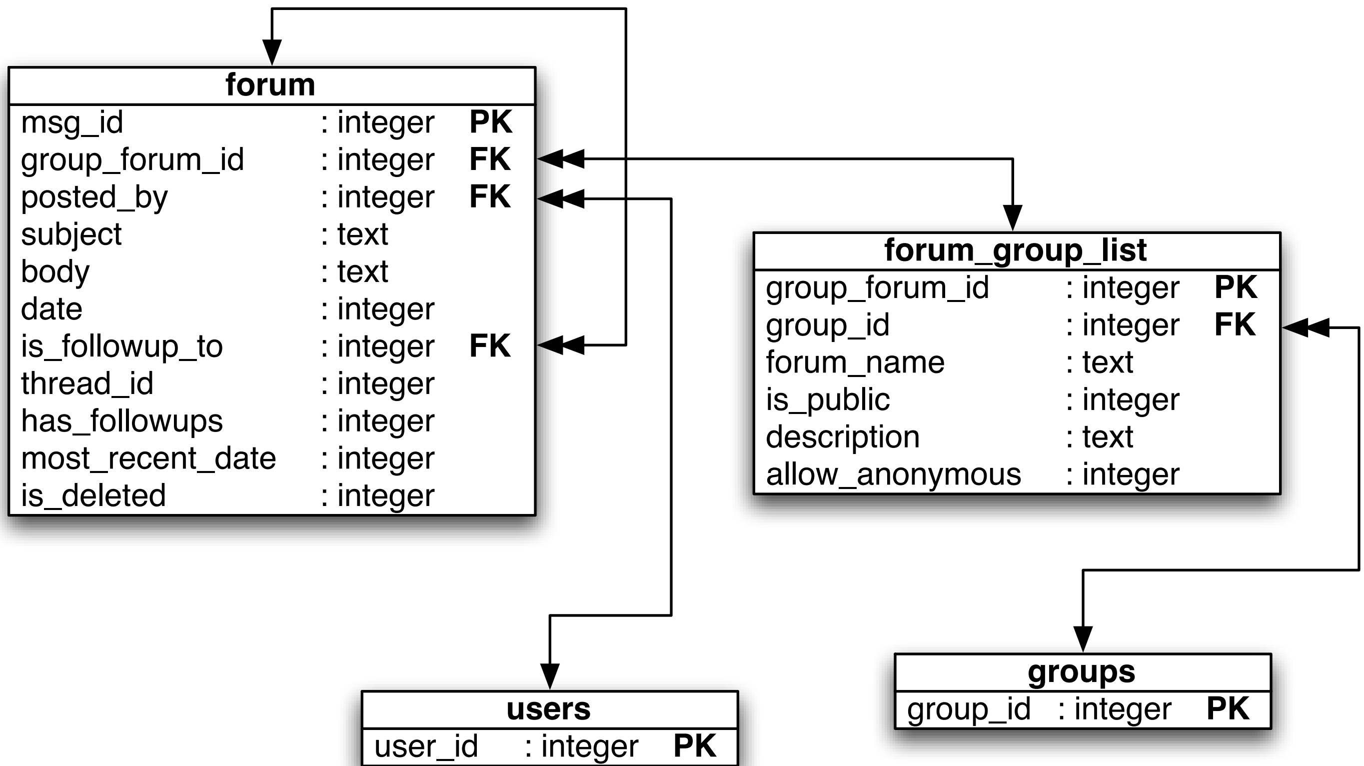

Figure 3: Schema for Forum Data in SourceForge Database

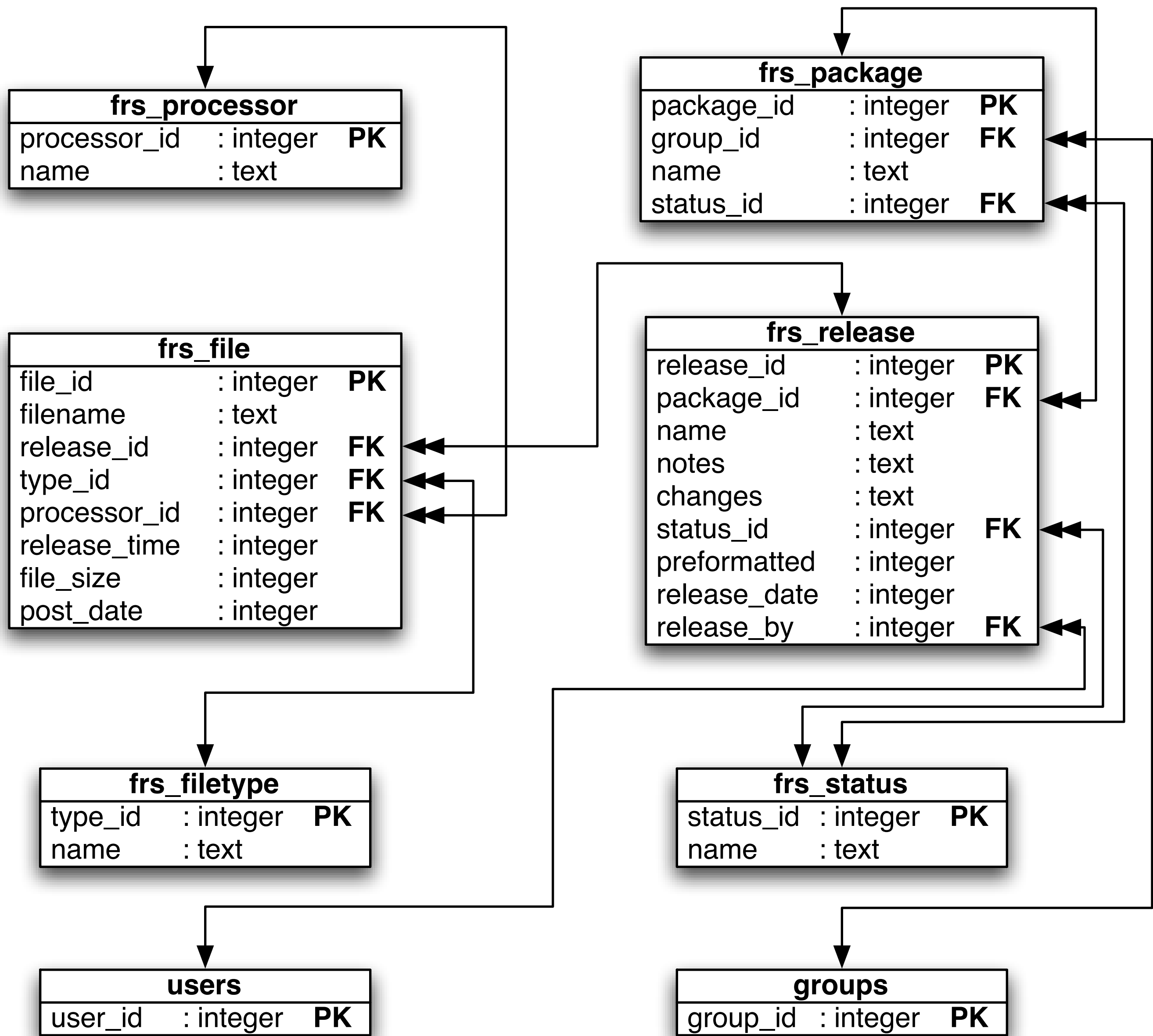

Figure 4: Schema for File Release Data in SourceForge Database

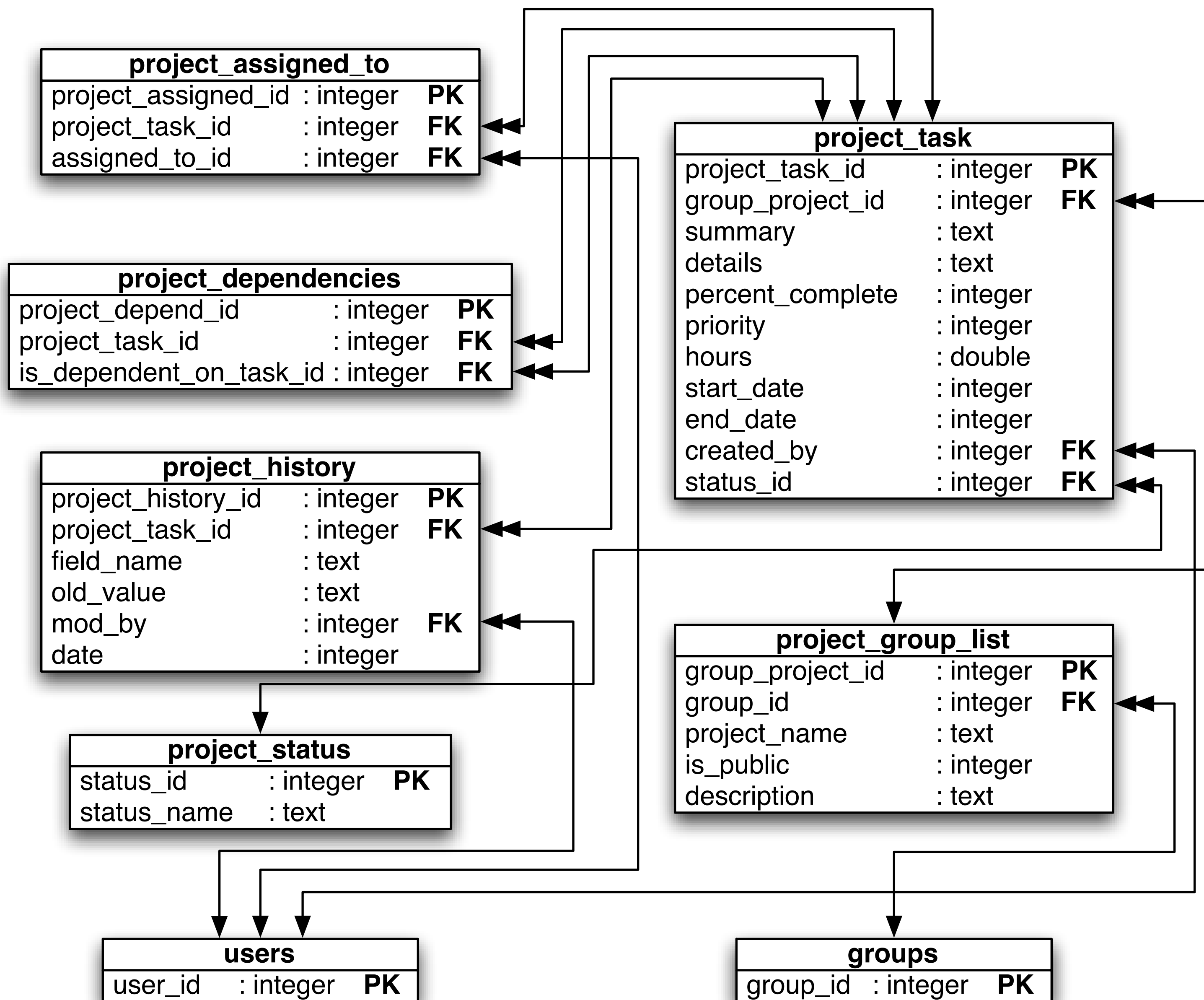

Figure 5: Schema for Project Task Data in SourceForge Database

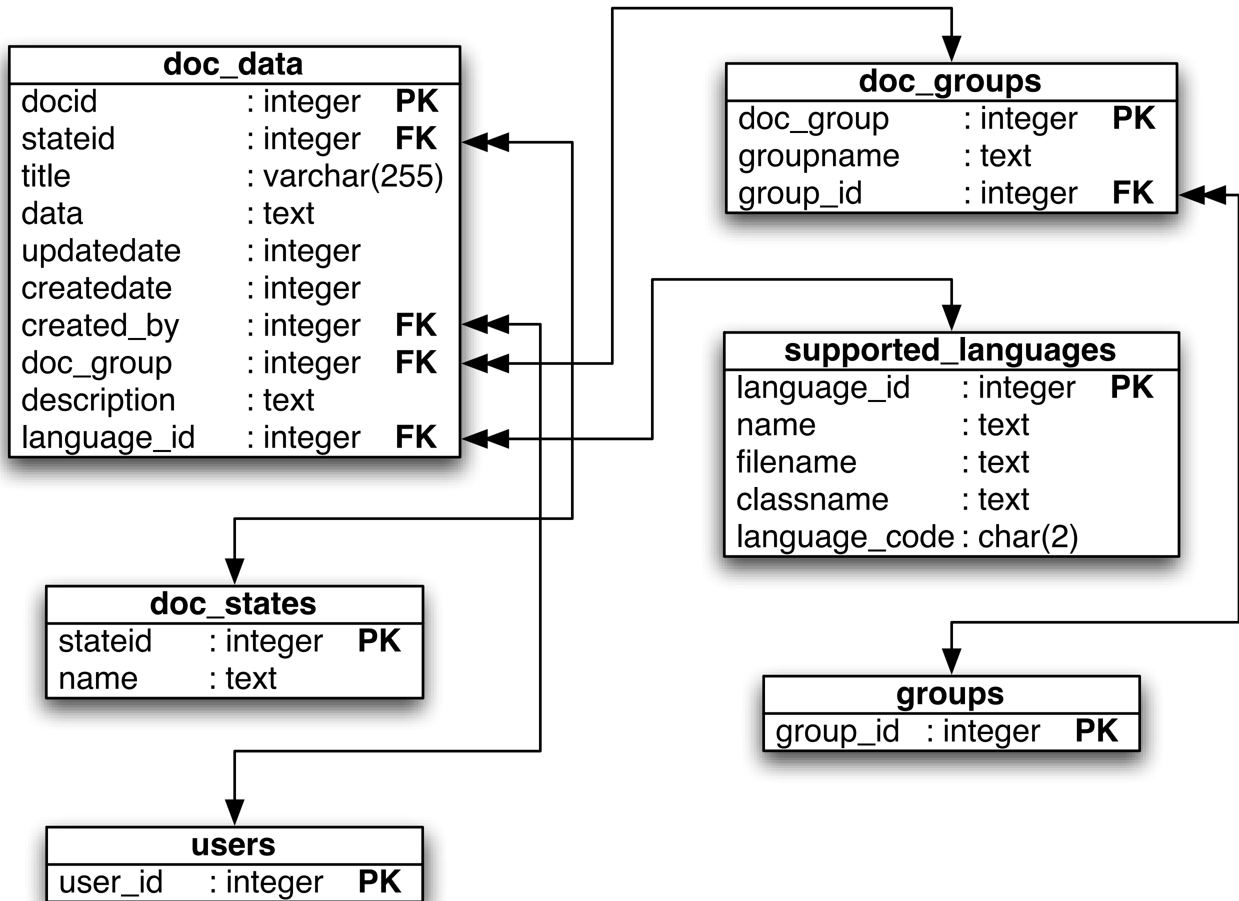

Figure 6: Schema for Documentation Data in SourceForge Database

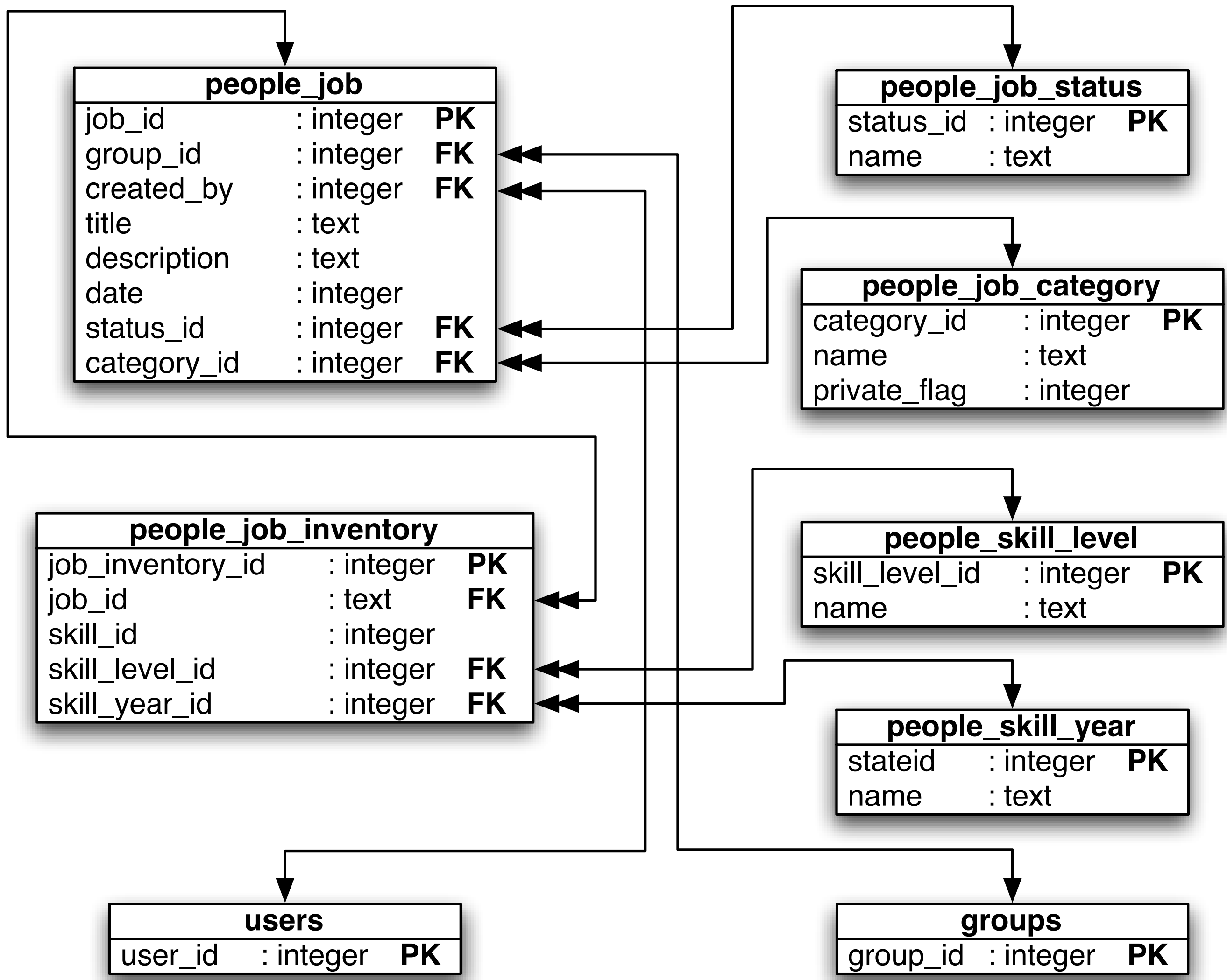

Figure 7: Schema for Project Job Data in SourceForge Database

Activity	Activity Type	Activity Description
1	submit bug	Person submits a new bug report.
2	assign bug	Bug report is assigned to person.
3	submit support request	Person submits a new support request.
4	assign support request	Support request is assigned to person.
5	submit patch	Person submits a new patch.
6	assign patch	Patch is assigned to person.
7	submit feature request	Person submits a new feature request.
8	assign feature request	Feature request is assigned to person.
9	submit todo	Person submits a new to-do item.
10	assign todo	To-do item is assigned to person.
11	submit other artifact	Person submits an artifact that is not one of the predefined categories of bug report, support request, patch, feature request, or to-do item.
12	assign other artifact	Uncategorized artifact is assigned to person.
13	new forum message	Person posts a new forum message.
14	followup forum message	Person posts a forum message that is a followup to an existing forum message.
15	modify project	Person makes an administrative modification to the project; the modification is uncategorized, but they are typically tasks like adding/removing members, changing permissions, updating project settings, etc.
16	file release	Person posts a new file release; this is typically associated with releasing a new version of the software to the public.
17	new project task	Person creates a new project task.
18	assigned project task	A project task is assigned to person.
19	modify project task	Person modifies an existing project task.
20	create document	Person creates a new document.
21	create people job	Person posts a new job; these are similar to help-wanted ads where a project is looking for somebody with particular skills.
22	checkout source code	Person checks out source code from CVS repository.
23	export source code	Person exports source code from CVS repository.
24	release source code	Person releases check out of source code from CVS repository.
25	tag source code	Person tags source code in the CVS repository with a label.
26	add source code file	Person adds a new source code file to the CVS repository.
27	remove source code file	Person removes a source code file from the CVS repository.
28	modify source code file	Person commits a source code modification to the CVS repository.
29	update source code	Person updates local checked out source code with any changes in CVS repository.

Table 1: List of Activity Types.

```
select a.artifact_id, b.group_id, b.name, a.assigned_to,
   a.submitted_by, a.open_date, a.close_date
   from artifact a, artifact_group_list b, groups c
   where c.group_id = b.group_id and b.group_artifact_id = a.group_artifact_id;
```

Figure 8: SQL query that joins artifact data together.

```
select id, project_id, name, assigned_by, submitted_to, open_date, processed
   from artifact_all where name = 'Bugs';

select id, project_id, name, assigned_by, submitted_to, open_date, processed
   from artifact_all where name = 'Support Requests';

select id, project_id, name, assigned_by, submitted_to, open_date, processed
   from artifact_all where name = 'Patches';

select id, project_id, name, assigned_by, submitted_to, open_date, processed
   from artifact_all where name = 'Feature Requests';

select id, project_id, name, assigned_by, submitted_to, open_date, processed
   from artifact_all where position('to' in lower(name)) != 0
   and position('do' in lower(name)) != 0;

select id, project_id, name, assigned_by, submitted_to, open_date, processed
   from artifact_all where processed = 0;
```

Figure 9: Set of SQL queries to extract different artifact types into separate activity types.

```
select a.msg_id, b.group_id, b.is_public, a.posted_by, a.date,
   a.most_recent_date,  a.is_followup_to
   from forum a, forum_group_list b
   where b.group_forum_id = a.group_forum_id;
```

Figure 10: SQL query to collect forum activity records.

```
select b.group_id, a.released_by, a.release_date
   from frs_release a, frs_package b
   where a.package_id = b.package_id;
```

Figure 11: SQL query to collect file release activity records.

```
select b.group_id, a.created_by, a.start_date
   from project_task a, project_group_list b
   where a.group_project_id = b.group_project_id;
```

Figure 12: SQL query for create project task activity records.

```
select c.group_id, a.mod_by, a.date
   from project_history a, project_task b, project_group_list c
   where a.project_task_id = b.project_task_id
   and b.group_project_id = c.group_project_id;
```

Figure 13: SQL query for modify project task activity records.

```
select c.group_id, a.assigned_to_id
   from project_assigned_to a, project_task b, project_group_list c
   where a.project_task_id = b.project_task_id
   and b.group_project_id = c.group_project_id;
```

Figure 14: SQL query for assigned project task activity records.

```
select b.group_id, a.created_by, a.start_date
   from project_task a, project_group_list b
   where a.group_project_id = b.group_project_id;
```

Figure 15: SQL query to collect documentation activity records.

```
select group_id, created_by, date
   from people_job;
```

Figure 16: SQL query to collect project job activity records.