Gitana: a SQL-based Git Repository Inspector

Valerio Cosentino, Javier Cánovas Izquierdo, Jordi Cabot

Gitana: a SQL-based Git Repository Inspector

Valerio Cosentino[1], Javier Luis Cánovas Izquierdo[1,2], Jordi Cabot[2,3]

[1] AtlanMod team (Inria, Mines Nantes, LINA). Nantes, France
[2] UOC. Barcelona, Spain
[3] ICREA. Barcelona, Spain
`valerio.cosentino@inria.fr, jcanovasi@uoc.edu, jordi.cabot@icrea.cat`

Abstract. Software development projects are notoriously complex and difficult to deal with. Several support tools such as issue tracking, code review and Source Control Management (SCM) systems have been introduced in the past decades to ease development activities. While such tools efficiently track the evolution of a given aspect of the project (e.g., bug reports), they provide just a partial view of the project and often lack of advanced querying mechanisms limiting themselves to command line or simple GUI support. This is particularly true for projects that rely on Git, the most popular SCM system today.
In this paper, we propose a conceptual schema for Git and an approach that, given a Git repository, exports its data to a relational database in order to (1) promote data integration with other existing SCM tools and (2) enable writing queries on Git data using standard SQL syntax. To ensure efficiency, our approach comes with an incremental propagation mechanism that refreshes the database content with the latest modifications. We have implemented our approach in Gitana, an open-source tool available on GitHub.

Keywords: Git, SQL, conceptual schema

1 Introduction

Software development projects are inherently complex due to the extensive collaboration and creative thinking involved [1]. In the last years, several tools have been created to cope with such complexity by providing specific support for the different development activities. Probably the most relevant are: Source Control Management (SCM) systems to manage code repositories [2] [3], storing and tracking the different source code versions; issue tracker systems [4] to provide support on maintenance and evolution activities such as reporting bugs and requesting new features; and code review tools to increase the quality of the final software product [5] by recording the communications between reviewers and code authors. While these tools efficiently track the evolution of a specific aspect of the software project, each provides just a partial view of it and usually comes with insufficient means (e.g., only command line support or other simple user interfaces) to perform any non-trivial query operation that could shed some light on important aspects of the project status.

This is particularly true for Git [6], that has become the most popular SCM system thanks to superior off-line capabilities, easier branch management and its promotion by most well-known Open Source Software (OSS) forges (e.g., GitHub and BitBucket) [7]. Despite the existence of several project management and monitoring tools built on top of Git, there is still a major lack of data integration efforts between them and they all fall short regarding the possibilities they offer for advanced query functionalities, thus forcing practitioners to resort to learning and using several off-the-shelf tools (e.g., [8], [9]), based on predefined queries on individual aspects of the project.

To overcome this situation, in this paper we propose a conceptual schema for Git and an approach that, given a Git repository, exports its data to a relational database derived from our conceptual schema. After the initial creation of the database, an incremental update mechanism will synchronize the database with the repository at any moment by considering only the modifications in the repository that took place after the last update.

Once in our database, we can easily integrate it with data coming from other tools that rely on a database infrastructure (e.g., GHTorrent [10], a scalable and offline mirror of GitHub; and Gerrie [11], a data and information crawler for the code review tool Gerrit). Thus, we are able to offer a shared place to perform cross-cutting analysis of a software development project including for instance the collaboration information (e.g., issues and pull requests) together with the Git (e.g., patches, file renamed) and code review data (e.g., review comments).

Furthermore, our proposed relational database can be exploited using the standard SQL language (and any other database analytical processing tool), thus enabling any practitioner familiar with the SQL language to easily inspect Git repositories according to her specific needs. For this same purpose, we have also predefined several views and stored procedures to simplify the computation of useful metrics and simulation of typical Git commands.

We have implemented our approach in *Gitana*, an open-source tool available on GitHub [12].

The remainder of the paper is organized as follows. Section 2 presents the motivation and the state of the art. Section 3 describes our approach while Section 4 discusses the application scenarios. Section 5 reports on the implementation details and evaluation conducted on five projects in GitHub. Finally, Section 6 ends the paper with conclusion and future work.

2 Motivation and State of the Art

Git repositories are typically explored via the Git command line. However, this approach is a complex and tedious task which requires developers to have a deep knowledge of Git and shell commands. Moreover, the integration of the queried information with other tools is very limited.

As an example, performing a query to obtain the number of deleted files in a Git repository requires to complement the Git command with shell commands to present the result in a comprehensible way. Listing 1.1 shows a possible way to perform this query. As can be seen, the Git command involves a few advanced parameters and the output must be digested with additional shell commands to filter (i.e., `grep` command) and present the desired information (i.e., `wc` command to count).

Listing 1.1. Number of deleted files via command line.

```
git log --diff-filter=D
        --summary
| grep 'delete mode'
| wc -l
```

Listing 1.2. Number of modifications on a file (named `FileName`) per user via command line.

```
git log --full-history
        --format=tformat:%a FileName
| gawk
 '{ count[$1]++ } END
  { for(j in count) printf "%s: %s changes\n",
    j, count[j]}'
```

When the aim is to perform a more detailed analysis (e.g., where grouping or pattern-matching is required), the definition of queries with the command line can quickly become an almost impossible task for non-experts. For instance, Listing 1.2 shows a query to obtain the number of modifications on a file per user. As can be seen, the query requires using the Git command option `log` together with some advanced parameters and other shell commands (see the use of `gawk` command) to digest, analyze and present the data.

This situation has not been solved by the proliferation of Git-based tools we have witnessed in the last years. This affects not only the day-to-day operations on Git-based projects but also any attempt to analyze Git to unveil interesting facts on the development process, team dynamics, etc., that could be useful to optimize the management and evolution of the project. Or, given the tight relationship between Git and code hosting platforms for open source projects, any effort aimed at mining Git repositories to extract common patterns for OSS development.

Previous works focused on extracting information from SCM systems to perform specific analysis (e.g., bug prediction, logical coupling detection, etc.), in particular most of them ([13], [14], [15], [16], [17], [18], [19]) target older SCM systems such as CVS and SVN, while only few focus on Git ([7], [20], [8], [9], [21]). More recent works have the goal of analyzing complete OSS forges such as GitHub or SourceForge ([10], [22]). Our approach belongs to the first axis (i.e., it's not linked to a specific code hosting platform) and tries to overcome different limitations of previous works. Such limitations are presented and compared with our proposal below (see Tab. 1).

– *Generality.* All previous works target very specific information goals, thus they extract and store only a portion of the SCM data. This partial view of the information makes very difficult to extend and integrate these tools with others targeting a different perspective. Instead, we propose a database which stores all SCM information (coarse and fine-grained).
– *Flexibility.* Many previous works such as [8], [9] and [19] do not allow ad-hoc queries, and Git itself makes it possible by only mixing Git commands together with shell scripting, as shown before. On the other hand, the works in [13], [15], [17], [18] and [16] allow the definition of ad-hoc queries limited to the portion of

	[13] [18]	[15]	[16]	[14] [20] [7]	[17]	[9]	[21] [8] [19]	Gitana	Git
Generality								x	x
Flexibility								x	
Incrementality								x	x
Exportability			x		x	x		x	
Extensibility	x	x	x		x			x	
Availability		x	x			x	x	x	x

Table 1. Comparison with previous works.

the SCM information they store. Our approach aims at performing better since it mirrors all information in the Git SCM and stores it in a RDB, thus it is flexible and complete enough to satisfy all user's needs for querying purposes.

– *Incrementality.* None of the previous works provides an incremental propagation mechanism to align the database content with the SCM's latest modifications. Thus, they have to be executed each time the repository changes, which hampers scalability when dealing with large repositories. Our approach includes an incremental propagation mechanism that makes it suitable for both small and large repositories.
– *Exportability.* Works by [17] and [16] provide an exporter from the data stored in the database to XML, however the XML structure and level of detail is unclear (e.g., it is not said whether the XML format is a mere representation of each database table). On the other hand, the approach presented in [9] does not rely on a database, but presents the data extracted from the SCM in HTML, XML and plain-text format. Currently, Gitana provides a JSON exporter that restructures the database information to make it available in other technologies beyond SQL.
– *Extensibility.* Previous works such as [13], [15], [17], [18] and [16] rely on a database that should be modified to add new sources of information, however they do not discuss any extension mechanism. On the contrary, our approach discusses possible integrations with other sources of information such as bug tracking systems and code review tools, as we show in this paper. Such integration can be easily achieved by connecting the concepts of *Developer*, *Commit* or *File* embedded in our model to their equivalent in other sources.
– *Availability.* Only [19], [15], [16], [8], [9] and [21] make their implementations available to download. However only [21] and [15] have been active in the past years. Gitana has been made available on GitHub at [12].

In short, to the best of our knowledge no previous tool or research effort has proposed a conceptual schema for Git with the intent of (1) promoting data integration with other Git-based tools and (2) enabling advanced query functionalities for Git.

3 Our Approach

We propose a conceptual schema for Git to facilitate data integration between existing Git-based tools and advanced query operations. This schema is materialized as a relational database, for which we have defined an extraction process that populates and keeps it up-to-date. Next we describe the conceptual schema, the corresponding database schema and the extraction process.

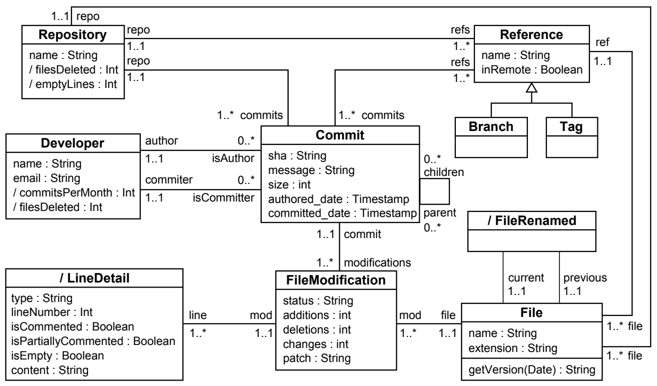

Fig. 1. Git conceptual schema.

3.1 Modeling Git

Git is a decentralized SCM system based on a master-less peer-to-peer replication where any replica of a given project can send or receive any information to or from any other replica. In Git, any developer can either create from scratch her own Git repository or obtain a copy (i.e., clone) of an existing one (i.e., remote). Although Git can theoretically work without a centralized repository, in practice there is usually a central repository that serves as the authoritative copy of the software project, thus it is what everyone fetches from and pushes to.

The structure of a Git *repository* is shown in the conceptual schema of Fig.1. A non-empty repository is organized following a tree structure where each node is represented by a *commit* to which *references* can be assigned. A commit is uniquely identified (i.e., using SHA) and contains a revision of the files within the repository reflecting the state of the project at a given point in time (i.e., snapshot).

In particular, a commit stores the differences between the files (*File*s and *FileModifications*) that changed between two revisions (the *patch* attribute in *FileModification* stores this raw information). It also contains information (i.e., name and email) regarding the corresponding author and committer (*Developer*), where the former is the one that did the change and the latter is the one that applied the change to the repository. Furthermore, a commit includes a reference to its parent commit(s) (*parent*). Generally, a commit has only one parent, that represents the previous state of the project; however, it can be parent-less (e.g., the commit originates the repository) or have multiple parents (when merging two or more branches).

Commits can be linked to different references, such as *branches* and *tags*. A *branch* represents a line of development that can be local if exists only in the cloned repository or remote if belongs to the remote repository (*inRemote*). The default branch is usually named *master* but new ones can be created to start new separated lines of development (e.g., to work on a new feature or to fix a bug). A branch can be also be merged with another one (e.g., to make a new release or when fixing a bug). A *tag* is a reference that can also be assigned to commits and it is generally used as marker for relevant events in the repository (e.g., releases, milestones).

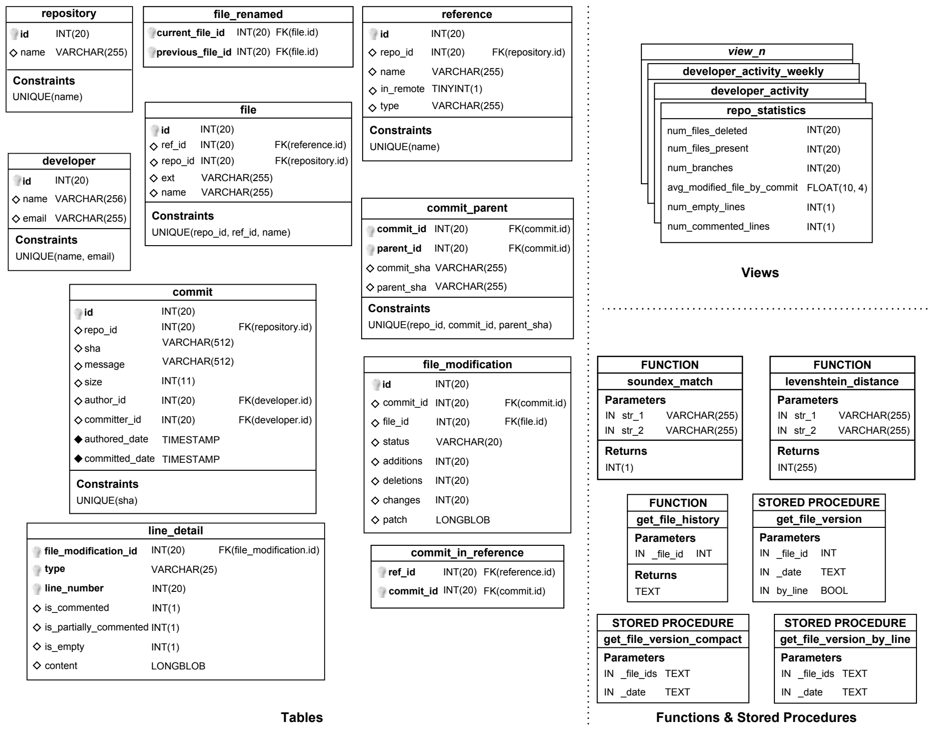

Fig. 2. Database Schema for Git.

The conceptual schema also includes some derived concepts and methods to facilitate the analysis of Git repositories by making explicit some information that is normally hidden or hard to query. This is the case for *LineDetail* and *FileRename* concepts, and the *getVersion* method of the *File* concept. *LineDetail* represents precise information regarding each line of a file modification, which includes the line number, its content and whether the line is (partially) commented or not. *FileRename* represents file rename actions occurred during the life-cycle of a software project, thus allowing tracking the whole life of a file in a repository. Finally, the *getVersion* method allows obtaining the version of a file at a particular timestamp.

Additionally, concepts in the schema can be enriched with some calculated metrics, expressed as derived attributes, to analyze the repository. As an example, we have added to the schema some derived attributes like *filesDeleted* and *emptyLines* providing some activity metrics of the repository.

3.2 A Database Schema for Git

The previous conceptual schema is materialized in the relational database schema of Fig. 2. In a nutshell, concepts/attributes in the conceptual schema are mapped into tables/columns in the database schema and associations are mapped into foreign keys (e.g., *author* association in *Commit* concept and *author_id* foreign key in *commit* table)

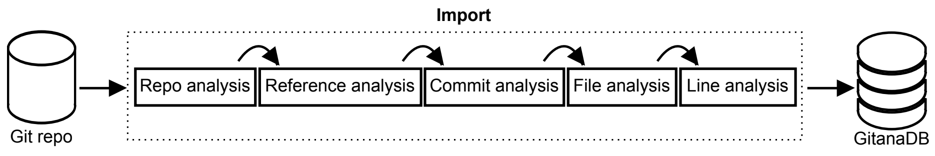

Fig. 3. Import process.

or new tables (e.g., *commit_parent*) depending on the cardinality of the association, following the typical translation strategies. Note that the *Branch-Tag* taxonomy has been mapped to a new attribute *type* in the table *reference*.

Additionally, several views have been created in the database to calculate the derived attributes in the conceptual schema (e.g., *repo_statistics* or *developer_activity* views). Auxiliary methods have been implemented as either functions or store procedures (see *get_file_version*). Full description of these views and methods can be found in the GitHub repository hosting the tool [12].

3.3 Extraction Process

We have defined an extraction process which interacts with a given Git repository in order to populate a database conforming to the schema previously described. The operations implemented by our approach support both the initial data loading from the SCM system to the database and later incrementally updating it with the latest SCM information. In the remainder of this section both operations are presented.

Initial Import Process An overview of the process to populate the database is shown in Fig. 3. It is composed of five steps, that respectively analyze and extract information concerning the repository, the references, the commits, the files and the file lines.

The first step, the *repository analysis*, adds to the table *repository* the name of the repository being analyzed. In the *reference analysis* step, branch and tag names are retrieved and used to fill the *reference* table together with the repository identifier.

In the *commit analysis* step, for each reference, all the corresponding commits, including common ancestors, are retrieved in chronological order (optionally, they can also be filtered by date if we don't want to import the full project history). For each commit, the names and emails of author and committer are stored in the table *developer*, then the corresponding author and developer identifiers are inserted together with the repository identifier and the commit information (e.g., SHA, message, etc.) in the table *commit*. Table *commit_in_reference* is used to relate commits with references, thus simplifying the retrieval of all commits in a given reference and the identification of commits shared between different references, while the table *commit_parent* is used to relate commits among each other, storing the relation between a commit and its corresponding parent(s).

In the *file analysis* step, for each commit, the differences (*patch*) between the previous and current versions of the files modified by the commit are retrieved. A patch can concern modifications (i.e., addition, deletion, changes) or renamings (i.e., changing its name or location) of a file. For each file involved in a patch, its name (including its path)

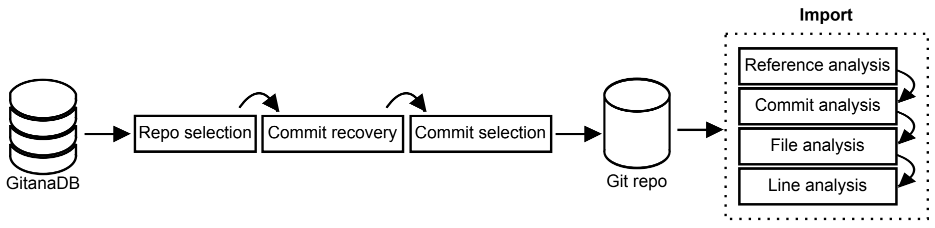

Fig. 4. Update process.

and extension as well as the identifiers of the repository and reference the file belongs to are stored in the table *file*. The identifier of the file is used to relate the file with its corresponding modifications and/or to its renamings. In particular, each modification on a file is stored in the table *file_modification*. It contains the identifiers of the related file and commit, the current status of the file at the time of that modification, the content of the patch and the number of additions, deletions and changes. On the other hand, if a file has been renamed, the previous and current identifiers of the files are stored in the table *file_renamed*.

Table *line_detail* is populated during the *line analysis* step. Such a step is in charge of analyzing and extracting the information concerning the individual file lines by using regular expressions on top of the patch information. In particular, the line number, the content of the line and the type of modification (i.e., addition, deletion) are retrieved together with information concerning whether the line is empty, commented or partially commented. The identification of comments is currently able to deal with line and block comments for the following languages: Python, Java, HTML, XML, SQL, JavaScript, C, C++, Scala, PHP, Ruby and Matlab.

Incremental Update Process The update process, shown in Fig. 4, keeps the information in the SCM system aligned with the one contained in the database. It is composed of three steps that connect the database with the Git repository plus a final *extraction* process to integrate the data not yet in the database.

The first step, *repository selection*, consists of selecting the name of a repository to update and retrieving the corresponding identifier stored in the table *repository*. In the *commit recovery* step, the repository identifier is used to collect the last SHA commit for each reference stored in the database by joining the information contained in the tables *repository*, *commit*, *commit_in_reference* and *reference*. Once the pairs {reference name, last SHA commit} are obtained, the *commit selection* step uses them to gather from the Git repository the set of new commits to be added to the database. Optionally, the retrieved commits can be filtered by retaining only those ones created before a certain date. In addition, the Git repository is queried also to collect those references that have no correspondence in the database.

Finally, some steps of the import process, previously presented, are reused to persist the new elements in the database, in particular, the *reference analysis* is started for each new reference found in the Git repository but not in the database, optionally together with a before date; while the *commit analysis* is launched for each set of new commits per reference.

Currently the materialized views in the database are recalculated each time the update process is triggered. Although this solution works properly with small and medium sized repositories, it may be inefficient for large repositories. In this sense, previous research efforts on incremental maintenance of materialized views (e.g., [23], [24], [25]) can be used to improve the efficiency of the update process.

4 Application Scenarios

In this section we present some application scenarios for the integration and advanced query functionalities provided by Gitana.

4.1 Integration

The schema presented in Fig. 2 can be integrated with other development-related data coming from tools that rely on a database infrastructure. Examples of such tools are most of issue tracking systems (e.g., BugZilla, Trac, Mantis) plus tools like GHTorrent [10], a scalable and offline mirror of GitHub; Gerrie [11], a data and information crawler for Gerrit, and Bicho [26], a tool that is able to parse different issue tracking systems (e.g., Launchpad, Jira, Allura).

Any of such tools embeds at least one concept that deals with *Commit*s, *File*s or *Developer*s. By leveraging on such concepts we can connect their database schemas with our Git schema. While the integration for files and commits is straightforward, since it involves a perfect match between file names and SHA identifiers; the integration for developers can only be semi-automatic and require the use of well-known identity matching/entity resolution algorithms (e.g., [27], [28]), as the developer can use different credentials (e.g., login or email) in the different tools she participates to.

As example, we illustrate how our approach can be integrated with GHTorrent and Gerrie. Figure 5 shows an excerpt of the main tables involved in the integration[4]. As can be seen, the concepts of *Developer*, *File* and *Commit* defined in our Git schema are similarly used in GHTorrent (tables *Users* and *Commits*) and Gerrie (tables *Gerrie_file*, *Gerrie_person*).

Integrating with GHTorrent provides a broader view of a GitHub project by combining the collaboration data and the Git SCM data, which GHTorrent does not cover at the moment. Thus, it opens up the possibility of writing queries that rely on both collaboration and code information (e.g., the most conflictive file lines according to the number of pull requests modifying them). Gerrie integration allows extending the analysis of code-reviews with fine-grained information from the Git SCM data (e.g., most influential developers in terms of changes in the files for a particular branch). Note that it is also possible to integrate the three tools, where Gitana may act as pivot representation. This scenario would provide a general view of the collaboration in the development process, for instance, to analyze the most conflictive pull requests (information provided by GHTorrent) in terms of code reviews (information provided by Gerrie).

[4] GHTorrent and Gerrie schemas are available at: http://ghtorrent.org/files/schema.png and http://gerrie.readthedocs.org/en/latest/database/#schema, respectively

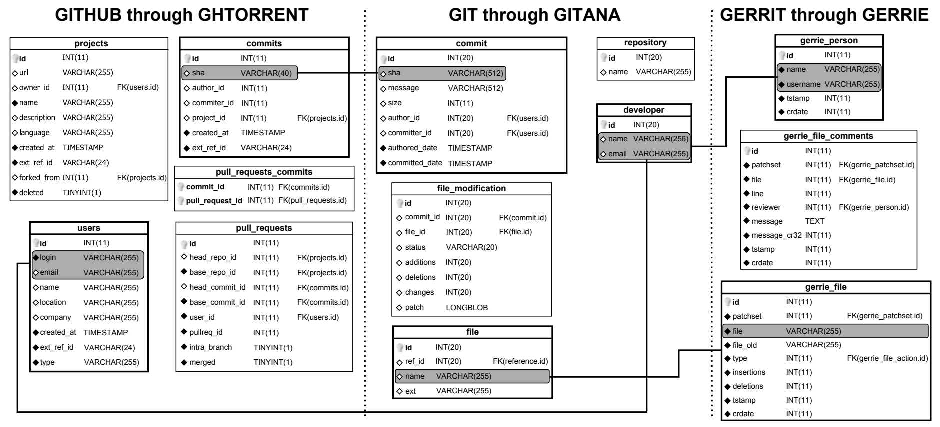

Fig. 5. GHTorrent and Gitana database schema excerpts and the main joining columns.

Listing 1.3. Number of deleted files via SQL.

```
SELECT
  COUNT(DISTINCT fm.file_id)
FROM
  file_modification fm
WHERE
  status = 'deleted';
```

In these examples, mappings are mostly one-to-one but it could happen that concepts are expressed with a different structure in the tools to be combined. In that case, previous findings on structural conflict resolution in the integration of Entity-Relationship schemas (i.e., [29], [30], [31]) should be used to deal with such schematic discrepancies.

4.2 Advanced Query Functionalities

Given the database representation of the SCM system, we can leverage on the plethora of tools and techniques existing in the database realm to perform advanced queries on the data. In this section we illustrate the advantages of this approach when compared to the traditional command line support.

For this, we show how we can easily rewrite the examples from Sect. 2 in SQL. Listings 1.3 and 1.4 show the SQL queries required to calculate the number of deleted files and the main modifications (i.e., additions, deletions and total) made by each developer in the repository, respectively (as done in Listings 1.1 and 1.2). As can be seen, instead of using a combination of command line and shell commands, with Gitana, pure SQL syntax suffices to get the information.

Additionally, our approach also includes some calculated information not directly available when using the Git command line. This can help developers to uncover valuable information from the repository. For instance, the derived concept *LineDetail* and its corresponding database table *line_detail* may help to discover who are the developers that comment the most the source code. Listing 1.5 shows a SQL query which calculates the number of files including comments per developer. In order to do so without

Listing 1.4. Number of modifications on a file (named `FileName`) per user via SQL.

```
SELECT
  d.name,
  count(distinct c.id) AS changes
FROM
  file f, file_modification fm,
  commit c, developer d
WHERE
  f.name = "FileName" AND f.id = fm.file_id AND
  fm.commit_id = c.id AND c.author_id = d.id AND
  f.ref_id = 1 /* a branch id */
GROUP BY
  d.id;
```

Listing 1.5. Number of files commented per developers via SQL.

```
SELECT
  d.name AS developer,
  count(distinct(fm.file_id)) as num_files
FROM
  line_detail ld, file_modification fm,
  commit c, developer d
WHERE
  ld.file_modification_id = fm.id AND
  fm.commit_id = c.id AND c.author_id = d.id AND
  (ld.is_commented = 1 OR
   ld.is_partially_commented = 1)
GROUP BY
  d.name;
```

Gitana, developers would need to come up themselves with the regular expressions to analyze the code.

Beyond pure SQL queries, we can also apply on our Git schema any existing ETL (Extract, Transform and Load) and OLAP (On-line Analytical Processing) technologies to perform multidimensional analysis for Git. Such a multidimensional analysis model can be personalized [32] to provide users with appropriate structures allowing them to intuitively analyze and understand the SCM information by reporting on current data, looking at historical data and trying to make predictions about future trends. Examples of possible analysis could be the evolution of the number of commits and files per unit of time (e.g., week, month, quarter) as well as the contribution activity of the developers in the project over time.

5 Tool support and Evaluation

Our method has been implemented in a tool called Gitana, available at [12]. Gitana relies on different technologies. The database import process has been implemented in Python 2.7.6 and rely on version 0.3.1 of GitPython[5], a library to interact with Git repositories. The generated database is by default stored in a MySQL server. The tool is launched via a simple GUI interface shown in Fig. 6.

[5] https://pypi.python.org/pypi/GitPython

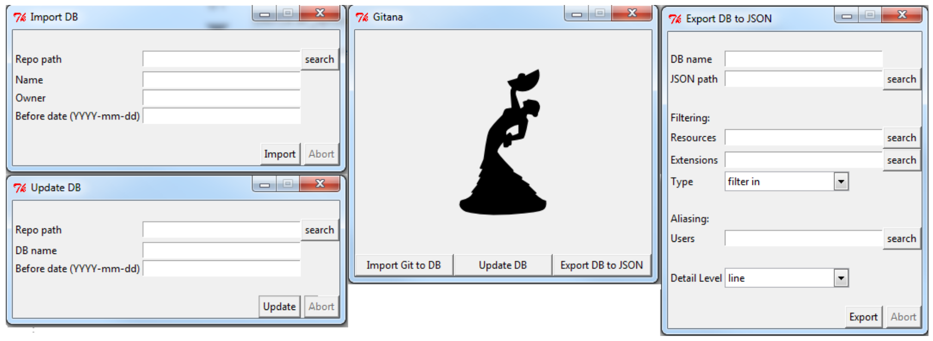

Fig. 6. Gitana GUI interface.

owner-repo	branches	tags	# commits in refs	extraction time
atlanmod-EMFtoCSP	1	-	67	5m37s
atlanmod-gila	2	-	289	8m47s
atlanmod-collaboro	3	3	1083	2h2m33s
octopress-octopress	1	38	5215	5h25m8s
reddit-reddit	2	-	10680	9h39m29s

Table 2. Evaluation on 5 open-source projects.

A JSON export component is also available. The JSON export process is composed of four steps: (1) repository selection; (2) file selection, which collects/ignores the files in the repository fulfilling some conditions (e.g., file extensions, inclusion in specific directories); (3) developer aggregation, which allows to merge together different user names corresponding to the same physical person or developers part of the same development sub-team and (4) file export, which collects the information for each file and generates the JSON data. The resulting JSON document follows a tree-based representation of the database where each entry is composed of the overall information (e.g., extension, current status and last modification) of a file, the commits that modified it, the list of changes at file level with the corresponding patches and the list of changes at line level. We believe the JSON support can facilitate the analysis of Git repositories by means of other technologies. For instance we described in [33] a website generator, relying on Gitana, to display the *bus factor*[6] of Git repositories.

Our extraction process has been evaluated on five open-source projects available on GitHub. Beyond validating that the extracted information was correct (either manually or, for some repositories, by discussing it with the project owners), the goal was to check the efficiency of the process. The results of the evaluation are shown in Tab. 2. We ran the evaluation on a 2.6 GHz Intel Core i7 processor with 8 GB of RAM. Note that even if the database import process can take some time (e.g., 2 hours for 1,000 commits), this only refers to the initial import. Once this phase is complete, the incremental mechanism takes over and minimizes the time for future imports.

[6] The bus factor of a project is typically defined as the number of key developers who would need to be incapacitated, i.e., hit by a bus, to make the project unable to continue

6 Conclusion

In this paper we have presented a conceptual schema for Git and how it can be used to export the data contained in a Git SCM system to a relational database in order to achieve two goals: (1) facilitate the integration with different tools built on top of Git and (2) enable advanced queries to inspect the repository, hard to achieve through the command line interface. Our method supports an incremental update of the Git data and has been implemented in the Gitana tool, freely available on GitHub at [12].

As further work, we would like to have a deeper integration of all kinds of SCM tools advancing in our idea of having one single central (database-oriented) shared access point for all the project information, enabling lots of interesting cross-cutting queries. Moreover, at the tool level, we would like to speed up the initial import phase by parallelizing the analysis of branches and tags in the repository. We are also interested in making more tunable the output of the JSON exporter, as currently the user is bound to the JSON predefined output structure.

References

1. Cockburn, A., Highsmith, J.: Agile software development: The people factor. Computer **34**(11) (2001) 131–133
2. Rochkind, M.J.: The Source Code Control System. Trans. Softw. Eng. (4) (1975) 364–370
3. O'Sullivan, B.: Making Sense of Revision-Control Systems. Comm. ACM **52**(9) (2009) 56–62
4. Serrano, N., Ciordia, I.: Bugzilla, itracker, and other bug trackers. Soft. **22**(2) (2005) 11–13
5. Kemerer, C.F., Paulk, M.C.: The impact of design and code reviews on software quality: An empirical study based on psp data. Trans. Softw. Eng. **35**(4) (2009) 534–550
6. Chacon, S., Hamano, J.C.: Pro Git. Volume 288. (2009)
7. Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., German, D.M., Devanbu, P.: The Promises and Perils of Mining Git. In: MSR. (2009) 1–10
8. Gitstats. http://gitstats.sourceforge.net/: (2007)
9. GitInspector. https://code.google.com/p/gitinspector/: (2012)
10. Gousios, G., Spinellis, D.: GHTorrent: Github's Data from a Firehose. In: MSR. (2012) 12–21
11. Gerrie. http://gerrie.readthedocs.org/en/latest/index.html: (2013)
12. Gitana website. https://github.com/SOM-Research/Gitana
13. Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: ICSM. (2003) 23–32
14. Zimmermann, T., Weißgerber, P.: Preprocessing cvs data for fine-grained analysis. In: MSR. (2004) 2–6
15. Robles, G., Koch, S., GonZÁlez-Barahona, J.M., Carlos, J.: Remote Analysis and Measurement of Libre Software Systems by means of the CVSAnalY Tool. In: RAMSS. (2004) 51–55
16. Draheim, D., Pekacki, L.: Process-centric analytical processing of version control data. In: IWPSE. (2003) 131–136
17. Robles, G., González-Barahona, J.M., Ghosh, R.A.: Gluetheos: Automating the Retrieval and Analysis of Data from Publicly Available Software Repositories. In: MSR. (2004) 28–31

18. Antoniol, G., Di Penta, M., Gall, H., Pinzger, M.: Towards the integration of versioning systems, bug reports and source code meta-models. Electron. Notes Theor. Comput. Sci. **127**(3) (2005) 87–99
19. Stephany, F., Mens, T., Gîrba, T.: Maispion: A Tool for Analysing and Visualising Open Source Software Developer Communities. In: ST. (2009) 50–57
20. Lee, H., Seo, B.K., Seo, E.: A Git Source Repository Analysis Tool Based on a Novel Branch-Oriented Approach. In: ICISA. (2013) 1–4
21. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: ICSE. (2013) 422–431
22. Williams, J.R., Di Ruscio, D., Matragkas, N., Di Rocco, J., Kolovos, D.S.: Models of oss project meta-information: a dataset of three forges. In: MSR. (2014) 408–411
23. Gupta, A., Mumick, I.S., et al.: Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Eng. Bull. **18**(2) (1995) 3–18
24. Staudt, M., Jarke, M.: Incremental maintenance of externally materialized views. In: VLDB. Volume 96. (1996) 3–6
25. Ross, K.A., Srivastava, D., Sudarshan, S.: Materialized view maintenance and integrity constraint checking: Trading space for time. In: SIGMOD Rec. Volume 25. (1996) 447–458
26. Gonzalez-Barahona, J.M., Izquierdo-Cortazar, D., Robles, G., del Castillo, A.: Analyzing gerrit code review parameters with bicho. ECEASST (2014)
27. Christen, P.: A comparison of personal name matching: Techniques and practical issues. In: ICDM. (2006) 290–294
28. Goeminne, M., Mens, T.: A comparison of identity merge algorithms for software repositories. Sci. Comput. Program. **78**(8) (2013) 971–986
29. Lee, M.L., Ling, T.W.: A methodology for structural conflict resolution in the integration of entity-relationship schemas. Knowl. Inf. Syst. **5**(2) (2003) 225–247
30. Chai, X., Sayyadian, M., Doan, A., Rosenthal, A., Seligman, L.: Analyzing and revising data integration schemas to improve their matchability. VLDB **1**(1) (2008) 773–784
31. Haas, L.M., Hentschel, M., Kossmann, D., Miller, R.J.: Schema and data: A holistic approach to mapping, resolution and fusion in information integration. In: ER conf. (2009) 27–40
32. Garrigós, I., Pardillo, J., Mazón, J.N., Trujillo, J.: A conceptual modeling approach for olap personalization. In: ER. (2009) 401–414
33. Cosentino, V., Cánovas Izquierdo, J.L., Cabot, J.: Assessing the Bus Factor of Git Repositories. In: SANER. (2015) 499–503