卒 業 研 究 報 告 書

題　目　　Gitリポジトリのマイニングにおける
クエリ言語の活用

指導教員　　水野　修　准教授

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　12122044

氏　　名　　原田 禎之

平成28年2月15日提出

Git リポジトリのマイニングにおける クエリ言語の活用

平成 28 年 2 月 15 日

12122044　原田 禎之

概　要

ソフトウェア工学におけるリポジトリマイニングでは，主に Git や Subversion 等のバージョン管理システムのリポジトリに存在するデータを分析することが必要となる．その際に，従来ではリポジトリマイニングを行う研究者が直接バージョン管理システムを扱い，データを処理してきた．

そのため，リポジトリマイニングに関する研究をする際には必ずといっていいほどバージョン管理システムの扱いに習熟する必要があるのみならず，バージョン管理システムの利用を行うためのプログラムやツールを研究者自身が実装する必要性があった．この作業は本来分析したい問題の本質とは関係が薄く，その実装等に関してかかるコストを削減することは有意義である．

本研究では，リポジトリマイニングを行う際のバージョン管理システムの扱いを容易にするために，現在広く利用されているバージョン管理システム Git を取り上げ，バージョン管理システムのコマンドを直接用いるのではなく，クエリ言語である SQL の SELECT 文を用いて検索を行うツールをコスト削減手法として提案するとともに，実装したプロトタイプモジュール “git-sql module” について評価を行った．

本研究により実装したプロトタイプを用いた評価では，リポジトリマイニングにおける著名なアルゴリズムに対する Git コマンドを用いた実装と提案手法を用いた実装を用意した比較実験を行った．実験の結果，計算にかかる時間的コストは増加してしまったものの，研究者の実装するマイニングのためのソースコード分量を大幅に削減することが可能であることを示した．

目　次

1. 緒言

ソフトウェア工学におけるリポジトリマイニングとは，ソフトウェア開発に関するデータが蓄えられたリポジトリから必要な情報を抽出するという行為である．リポジトリマイニングが対象とするリポジトリには様々な種類があるが，本研究ではソフトウェア開発時の版管理データの蓄積するバージョン管理システムのリポジトリのうち，Git リポジトリを扱うものを対象とする．

従来，ソフトウェア工学における Git リポジトリへのマイニング活動では，Git コマンドを直接発行することにより結果を取得し，その結果を元にデータ処理を行うという手順が主に用いられてきた．そのため，リポジトリマイニングを行う研究者はソフトウェア工学に直接は関係のない Git コマンドの扱いにまで習熟する必要があり，Git コマンドを用いる実装を行うことに研究リソースを割かざるをえず，そのコストは可能ならば削減したいものとされてきた．

そこで，本研究では，Git リポジトリを検索するために，一般的にクエリ言語として用いられている SQL を利用するツールをリポジトリマイニングに用いることを提案する．

また，クエリ言語の利用がリポジトリマイニングにおいて有効であることを実証するため，SQL の SELECT 文を用いて Git の版管理データを検索する機能を実装したプロトタイプモジュール “git-sql module” を実装し，評価した．評価を行う際にはリポジトリマイニングにおいて有名なアルゴリズムである SZZ アルゴリズムを実装して既存の実装と比較することで，リポジトリマイニングの際のコストがどのように変化したかについて述べる．

本報告の以降の構成は次に述べるとおりである．第 2 章では，本研究で用いる SQL や Git の用語の説明を行う．第 3 章では，本研究でプロトタイプとして実装したシステムの概要について説明を行う．第 4 章では実装したプロトタイプについて行った評価について説明する．第 5 章では本研究についての考察を述べる．最後に，第 6 章では，本研究のまとめ及び今後の課題を述べる．

2. 用語

2.1 バージョン管理システム

バージョン管理システム（バージョン管理システム）とはコンピュータ上で作成，編集されるファイル，主にソフトウェア開発におけるソースコードの変更履歴を管理するためのシステム．本研究では Git を対象とする．

2.1.1 Git [1]

Git は Linus Benedict Torvalds 氏によって Linux カーネルの版管理を行うために開発された．

バージョン管理システムの中でも分散型バージョン管理システムに分類され，バージョン管理システム，ローカルリポジトリを作成，その中で作業を行う．

(1) Git を構成するオブジェクト [4]

Git は様々なオブジェクトにより構成されるが，本研究で扱うのは以下のデータとする．

ユーザ バージョン管理システムを利用するユーザ.

リポジトリ バージョン管理システムで扱う対象．プロジェクト本体.

コミット Git に管理された変更履歴の最小単位.

ファイル（ツリー） 版管理の対象となるファイル．ツリーにより管理される．

(2) 主な Git の用語とコマンド

Git に関係する主な用語（コマンド）のうち，本研究に関係するものは以下のとおりである．

ブランチ 特定のコミットを指すポインタ.

clone リモートのリポジトリを元にローカルリポジトリを作成するコマンド.

merge 特定のブランチに別のブランチの内容を取り込むコマンド.

pull　リモートリポジトリの内容をローカルへ反映し，マージを行うコマンド.

push　ローカルのリポジトリの変更をリモートへ送信するコマンド.

diff　2 つのコミット間の変更情報を表示するコマンド.

2.2　クエリ言語

クエリ言語（問い合わせ言語，Query language: QL）とは，コンピュータ中のデータに対して問い合わせを行うためのコンピュータ言語であり，本研究では特にリレーショナルデータベースに対するクエリ言語である SQL を用いる.

2.2.1　SQL [2]

SQL はリレーショナルデータベース管理システムにおいて，データの操作や定義を行うためのデータベース言語（クエリ言語）である.

SQL には大きく分けてデータ定義言語，データ操作言語，データ制御言語が存在する．それらの役割と具体的にどのような文が分類されているかを以下に示す.

データ定義言語

関係データベースの構造を定義する際に用いる言語である．主として以下の 4 文から成る.

CREATE 文　新しいデータベースやテーブルなどを作成する.

DROP 文　既存のデータベースやテーブルなどを削除する.

ALTER 文　既存のデータベースオブジェクトに対する変更.

TRUNCATE 文　既存のテーブルからのデータの不可逆的削除.

データ操作言語

データベースに対してデータの検索，新規登録，削除，更新を行うための言語である．主として以下の 4 文から成る.

INSERT 文　行データ，又は表データをデータベースへ挿入する際に用いる.

UPDATE 文　表中のデータを更新する際に用いる.

DELETE 文　表中から特定のデータを削除する際に用いる.

SELECT 文　表データに対し検索を行い，結果集合を取り出す際に用いる．

SELECT 文では “SELECT 取得したい列名 FROM 表名 WHERE 条件式 ” の形でデータベースの表内を検索することができる．条件式には AND や OR といった論理演算子の他に，入れ子式に，別の SELECT 文により検索を行った結果集合を用いて更なる検索を行う副問い合わせも可能である．

本研究ではこの SELECT 文に着目する．

データ制御言語

データベースにおいて，アクセス制御を行うための言語である．主として以下の 2 文から成る．

GRANT 文　データベース利用者へ権限を付与する．

REVOKE 文　データベース利用者の権限を剥奪する．

3. git-sql module

3.1 設計

3.1.1 概要

本研究で設計及び実装した git-sql module は，1 つのデータベースを 1 つのファイルとして管理することのできる SQLite [5] を用いて，任意の Git リポジトリのデータを元に SQLite データベースを構築することを考える．

本ツールは入力に検索対象としたいリポジトリ，SQL 文，出力先ファイル名，キャッシュ利用の可否などのオプションをとる Python2.7 スクリプト，及び Python2.7 モジュールとして設計した．

処理の概要は図 3.1 に示すように，以下とする．

1. Git のリポジトリから，コミットのログやユーザ情報，ファイル情報を抽出する．
2. 抽出した情報に基づきキャッシュとなる SQLite データベースを作成する．
3. 入力された SELECT 文を作成したキャッシュデータベースへ発行し，クエリ結果を生成する．

3.1.2 データベースの構成

まず，Git リポジトリの中のデータのうち，リレーショナルデータベースにおける “データベース” とするもの及び “テーブル” とするものを決定する．

次に，策定した “テーブル” の構成要素となる “スキーマ” とするデータを決定する．

本研究では，“データベース” とするものを一つの Git リポジトリとし，“テーブル” とするもの及びそれらを構成する “スキーマ” は以下のとおりとした．

- commits

  コミットの情報を扱うテーブル．

  - id INTEGER PRIMARY KEY AUTOINCREMENT

    データベース中で一意識別可能なコミットの ID
  - sha VERCHAR(255)

    コミットの SHA-1 ハッシュ

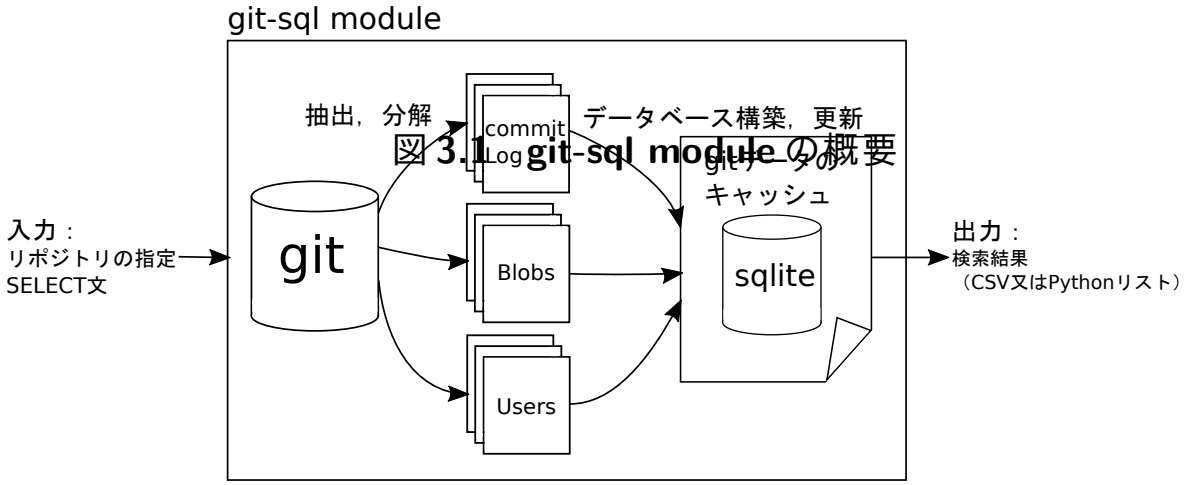

図 3.1 git-sql module の概要

- message TEXT

  コミットメッセージ

- files INTEGER

  コミットで変更されたファイルの数

- lines INTEGER

  コミットで変更された行数

- insertions INTEGER

  コミットで変更された行数のうち，挿入行数

- deletions INTEGER

  コミットで変更された行数のうち，削除行数

- author_id INTEGER

  コミットの著者のユーザ ID

- committer_id INTEGER

  コミットを行った人のユーザ ID

- committed_date TIMESTAMP

  コミットの日付

- files

  ファイルの情報のうち変更履歴以外を扱うテーブル．

  - id INTEGER PRIMARY KEY AUTOINCREMENT

    データベース中で一意識別可能な ID

  - path TEXT UNIQUE

    ファイルのフルパス

- users

  ユーザ情報を扱うテーブル．

  - id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT

    データベース中で一意識別可能なユーザ ID

  - name VERCHAR(255) DEFAULT NULL

    ユーザの名前（author 名等）

  - email VERCHAR(255) DEFAULT NULL

ユーザの名前（author 名等）

- tz_offset INTEGER

  ユーザのタイムゾーン情報

- commit_files

  コミットに含まれているファイルの詳細な変更情報を扱うテーブル．

  - commit_id INTEGER NOT NULL

    ファイルの所属するコミットの ID

  - file_id INTEGER NOT NULL

    ファイルの ID

  - sha VERCHAR(255)

    更新後のファイルの SHA-1 ハッシュ

  - parent_sha VERCHAR(255)

    更新前のファイルの SHA-1 ハッシュ

  - new_file BOOLEAN

    該当ファイルが新たに追加されたファイルであるか否か

  - deleted_file BOOLEAN

    該当ファイルが削除されたファイルであるか否か

  - deletions INTEGER

    該当ファイルの変更のうち，削除行数

  - insertions INTEGER

    該当ファイルの変更のうち，挿入行数

  - diff TEXT

    該当ファイルの変更全文（git diff コマンドの結果）

- commit_parents

  コミットの親子関係を扱うテーブル．

  - commit_id INTEGER NOT NULL

    親子関係の，子のコミットの ID

  - parent_id INTEGER NOT NULL

    親子関係の，親のコミットの ID

3.2 実装

本研究での実装には先述したとおり，Python2.7 [6] を用いた．他に外部依存モジュールとして GitPython [7] を利用した．これは Git のコマンドを Python スクリプトから利用することを容易にするモジュールである．

実装アルゴリズムの概要は以下のとおりである．

1. 指定されたリポジトリがリモートリポジトリであれば，リモートリポジトリをローカルへクローンする．
2. ローカルリポジトリからコミットのリストを取得する．
3. コミットリストの要素それぞれについて以下の処理を行う
   (a) キャッシュデータベースにレコードが存在すれば，以下の処理を飛ばして次のコミット要素へ移る．
   (b) コミットの author，committer について，user テーブルに要素が存在しなければ挿入を行う．それぞれ，user テーブルの ID を取得しておく．
   (c) コミットの要素について，commits テーブル及び commit_parents テーブルへのデータ挿入を行う
   (d) コミットに含まれるファイルについて，以下の処理を行う．
      i. files テーブルにファイルのパス情報が存在しなければ挿入する．同時に ID を取得する．
      ii. ファイルの更新情報について，commit_files テーブルへの挿入を行う．
4. 格納したデータベースキャッシュファイルへの検索を行い，結果を返す．

正確性を維持しつつ処理速度を確保するため，以下の点に留意し，実装を行った．

- 毎回データベースを構築していては時間がかかりすぎるため，可能な限りキャッシュを再利用する．
- 処理中の SQLite データベースのテンポラリをメインメモリ上に展開した．
- クエリを発行する際にかかる時間コストが大きいため，なるべくクエリの反復を避けるため，一度のクエリで多くの情報を扱う方針とした．

3.3 利用方法

本モジュールの利用については，コマンドラインで引数を指定する形で実行するか，Python2 モジュールとして import することにより利用するかの二種類の方法を提供する．

コマンドラインによる実行では出力はクエリ結果を表形式に整形した状態での標準出力への出力，または指定した CSV ファイルへのクエリ結果の出力に対応する．コマンドの書式はソースコード 3.1 に詳述する．

Python2 モジュールとしての実行では，gitsql モジュールとして import することにより以下の関数が利用可能となる．

- prepare(repo, cleanFlag, mode)

  Git リポジトリからデータベースを準備する関数である．クエリを行う前段階として，必要なオプションを指定して実行される．引数の説明は以下のとおりである．戻り値はデータベースの準備（構築，またはキャッシュへの接続）が成功すれば True を，失敗すれば False を返す．

  - repo

    文字列でリポジトリの URI もしくはパスを指定する．
  - cleanFlag

    True か False をとる．True を指定するとキャッシュを必ずクリアする．
  - mode

    データベース化が必要な要素を指定するモードを 0〜4 の数値で指定する．数値が大きくなればなるほどデータベース化が簡易になり，かかる時間が少なくなる．(実験的)

- query(*args, **kwargs)

  データベースへのクエリを行う．引数は Python の sqlite3 モジュールと同じ書式である．

  戻り値は Python のリスト形式で，第一要素に該当列の要素名をとる．第二要素以降はクエリ結果が格納される．

クエリ関数の戻り値として Python のリストによりクエリ結果を出力する．

ソースコード 3.1　git-sql module のコマンド書式

```
1  gitsql.py --sql "SELECT␣*␣FROM␣...." --repo "https://hogehoge" --output "hoge.csv"
       --mode MODE --cacheclear
2  --sql: SQL sentence (required)
3  --repo: Git repository URI or Directory (required)
4  --output: Output CSV file
5  --mode: Select DB create mode
6    0 create full DB (Default)
7    1 without blobs
8    2 without users
9    3 without blobs and users
10   4 without creating DB (only use cache)
11 --cacheclear: Clear DB cache
```

具体的な利用例は例えば，ソースコード 3.2，3.3 に記述したとおりとなる．これらの記述では本モジュールを利用してリポジトリ “/hoge/testrepo/” のコミットから追加行数が 100000 行を超えるものを抽出するために必要なものである．

ソースコード **3.2**　コマンドラインからの利用例

```
python gitsql.py \
--repo "/hoge/testrepo/" \
--sql "SELECT␣sha␣FROM␣commits␣WHERE␣insertions␣>␣100000;"
```

ソースコード **3.3**　**Python** スクリプトからの利用例

```
import gitsql
gitsql.prepare('''/hoge/testrepo/''', True, 0)
result = gitsql.query('''SELECT sha FROM commits WHERE insertions > 100000;''')
```

4. 評価

4.1 評価手法

実装した “git-sql module” を評価するために，本実験では簡単なリポジトリマイニングとして有名なアルゴリズムである，SZZ アルゴリズムを “git-sql module” を用いて実装し，既存手法による実装と比較した．

4.2 SZZ アルゴリズム [3]

4.2.1 概要

SZZ アルゴリズムはバージョン管理システムとバグ管理システムの情報を結びつけることで不具合の混入したモジュール，及び不具合混入が起きたコミット，不具合が修正されたコミットを自動で判別するアルゴリズムである．

これは，バージョン管理システムには存在しない不具合情報をバグ管理システムで補ってやるというアイディアである．

基本となるアルゴリズムは以下のとおりである．

1. バグ管理システムからバグの ID，及び報告された日付（Open date）とバグが修正された日付（Close date）のデータを取得する．
2. 取得したバグ ID について，Open date から Close date の間のコミット群のうち，コミットメッセージに “FIXED” や “RESOLVED” といった語句とともにバグ ID が出現したものをバグ修正コミットとして抽出する．
3. 抽出したバグ修正コミットそれぞれにおいて変更があったファイルを，不具合混入ファイルの候補として抽出する．
4. 不具合混入ファイルの候補のうち，Open date からバグ修正コミットまでの間に一度も変更されたことがないファイルを不具合混入ファイルとして特定する．
5. それぞれの不具合混入ファイルについて，それらが Open date 以前に変更された最後のコミットをバグ混入コミットとして抽出する．

4.2.2 扱うデータ

SZZ アルゴリズムは前述したとおり，過去のバグ情報を用いてバージョン管理システムのリポジトリにおけるバグの存在範囲を特定するものである．そのため，入力としてバグ管理システムのバグ情報のリスト，及びバージョン管理システムの履歴（Git の場合はコミットログやファイル情報）を扱い，それらを紐付けたデータを出力する．

特に，本研究で評価のために実装した SZZ アルゴリズムでは，入力にはバグ管理システム Bugzilla からエクスポートした CSV データ及び Git のリポジトリを用い，Git リポジトリのタグとして，又は CSV 形式としてバグの混入/修正コミット情報を出力する．

4.3 評価手順

4.3.1 git-sql module を用いた SZZ アルゴリズムの実装

SZZ アルゴリズムの概要については 4.2 章に示したとおりである．本研究では，SZZ アルゴリズムの概要に則って以下のとおりに評価用のタスクとしての SZZ アルゴリズムを実装した．

1. バグ管理システムから取得したバグリストを読み込む．
2. git-sql module の初期化を行うことにより，該当リポジトリへの接続を行う.（必要ならばデータベースキャッシュを作成する）
3. リスト中のバグデータのそれぞれについて，不具合の修正されたコミット “FIX_COMMIT” を特定するためにソースコード 4.1 に示した SELECT 文を生成，実行して結果を取得する．
4. 取得した FIX_COMMIT の ID について，該当のコミットで修正されたバグを含むファイルを抽出するため，ソースコード 4.2 に示した SELECT 文を生成，実行して結果を取得する．
5. バグを含むファイルとして抽出されたファイルそれぞれについて，そのバグが混入したコミットを特定するため，ソースコード 4.3 に示した SELECT 文を生成，実行して結果を取得する．

ソースコード 4.1　バグ修正 commit の特定

```
SELECT sha, id FROM commits
  WHERE (committed_date >= '該当バグのopen_date'
      AND committed_date <= '該当バグのclose_date')
    AND (message LIKE '%fix%バグID%' OR message LIKE '%resolve%バグID%');
```

ソースコード 4.2　不具合ファイルの抽出

```
SELECT id, path FROM files WHERE id IN
  (SELECT file_id FROM commit_files
    WHERE commit_id = 'FIX_COMMITのID' AND commit_id NOT IN
      (SELECT id FROM commits
        WHERE (committed_date >= '該当バグのopen_date'
          AND committed_date <
            (SELECT committed_date FROM commits WHERE id = 'FIX_COMMITのID')
        )
      )
  );
```

ソースコード 4.3　バグ混入 commit の特定

```
SELECT sha, committed_date FROM commits
  WHERE committed_date <= '該当バグのopen_date'
    AND id IN (SELECT commit_id FROM commit_files WHERE file_id = '不具合ファイルのID')
  ORDER BY committed_date DESC
  LIMIT 1 ;
```

4.3.2 実装した SZZ アルゴリズムの適用対象

(1) 適用対象のリポジトリ

本研究で実装した SZZ アルゴリズムを評価するために，本研究ではオープンソースプロジェクトの BIRT プロジェクト [8]（eclipse を利用した帳票管理システム）のリポジトリ [9] を対象とした．

BIRT プロジェクトのリポジトリは 2016 年 1 月 22 日時点で総コミット数 31541，管理されたことのある総ファイル数 20702 となっており，評価実験のために適切な規模のリポジトリであると考える．

(2) 適用対象のバグデータ

適用対象の BIRT リポジトリに対応するバグデータとして，Bugzilla のバグデータ [10] をもとに作成したデータを利用した．

本研究では，2005 年 1 月 7 日に Open されたものから，2010 年 6 月 25 日に Close したものまで，19787 件のバグ情報を利用した．

4.3.3 評価に用いる客観指標

本研究で用いる評価指標は，リポジトリマイニングに関する同じタスクを実行することを目的に実装したプログラムについて，タスクの処理時間及び実装に必要なコード分量の 2 点とする．

4.3.4 評価環境

タスクの処理時間に関して測定を行った環境は，同一ホストマシン上の Oracle VM VirtualBox 上で動作する Ubuntu15.10 で，ホストマシンのプロセッサ，論理 8 コアのうち論理 4 コアを割り当てた．

なお，計測の際は極力他の負荷を排除するために深夜などを利用した．

4.3.5 比較対象のプログラムについて

今回自作の git-sql module を利用した SZZ アルゴリズムと比較するために，クエリ言語を用いずに逐次 Git コマンドを発行することによって SZZ アルゴリズムを実行

するツールである，szz_tool [11] というソフトウェアとの比較実験を行った．szz_tool はスクリプト言語 Perl [12] を用いて実装されている．

4.4 評価結果

評価の概要は表 4.1 にまとめたとおりである．

実行にかかった時間について，git-sql module を利用した SZZ アルゴリズムは従来手法と比べ，SQLite のデータベースキャッシュを使用せず，リポジトリからの構築を行った場合では 561.0 分かかるという結果となり，328.6 分で処理が完了した従来手法と比べ 1.7 倍処理時間がかかってしまう結果となった．

一方，データベースキャッシュを使用した場合，すなわち同じアルゴリズムを既存のデータベースを用い，差分のデータのみデータベースの構築を行って繰り返しおこなった場合は 129.4 分で処理が完了し，従来手法の 0.23 倍の処理時間で済むということがわかった．

また，同じアルゴリズムの実装のために掛かったソースコードの量は従来手法では 986 行，26,226byte であったのに対し，git-sql module を利用した場合では 112 行，4,512byte となり，行数で約 0.11 倍，ファイルサイズで約 0.17 倍のソースコード量で SZZ アルゴリズムの実現が可能であった．双方ともに評価すべきであるのはアルゴリズムの実装コストであるため，外部ライブラリ（git-sql module 含む）のソースコード量は考慮していない．

表 4.1　従来手法と git-sql module 利用の SZZ アルゴリズムの比較

		処理時間	ソースコード量	
従来手法（Perl）		328.6 分	986 行	26,226byte
git-sql module（Python2）	キャッシュ使用無し	561.0 分	112 行	4,512byte
	キャッシュ使用	129.4 分		

5. 考察

本章では，本研究で行った比較実験の妥当性について考察を述べる．

5.1 評価指標の妥当性

本研究で用いた評価指標はタスクの実行速度とソースコード量（コード行数，ファイルサイズ）である．

これらの指標は同一言語での実装，同じ実装者による実装であれば比較的妥当な指標となりうる．しかし，今回の実装では従来手法は Perl を用い，提案手法では Python を用いており，更に実装を行った開発者が異なっており，単純に比較できないのではないかという懸念がある．

まず開発言語が違う点について，Programming languages ranked by expressiveness [13] によると Python と Perl の間には，行あたりの表現力についてどちらか一方が優位であるという違いは少ないという結果が得られている．これにより，コード分量を指標として用いることの開発言語の差異による影響は，今回のケースでは無視できると考える．

また，本実験では敢えてソースコード中のコメントを残したままソースコード量を計測した．これはコメントを残す必要性なども同じタスクの実装にかかるコストとして評価できると考えたためである．

次に，開発者が異なっている点について，本研究で比較対象としたアルゴリズムはソフトウェア工学において用いられるリポジトリマイニングに関係するものである．そのため，既存手法による実装もソフトウェア工学に携わる研究者によって実装されており，開発者の差がコード分量に及ぼす影響は比較的軽微なものであると考える．

また，開発者の違いが実装方針に影響し，タスクの実行速度に影響をあたえる可能性が考えられるが．SZZ アルゴリズムはもともとある程度時間計算量の大きいアルゴリズムであり，実装する上で最低限の高速化は必要であると考える．そのため，従来手法でもいたずらに速度を犠牲にする実装がなされていたとは考えづらく，速度比較の上でも開発者が異なることによる影響は本研究で提案した手法の与える影

響より十分に小さいと考えられる．更に，比較対象の従来ツールがマルチスレッディングに対応していなかったため，git-sql module を用いた実装でもマルチスレッディングを利用せず，CPU リソースのうちシングルスレッドのみで利用可能な範囲を利用するにとどめた．

これらより，本実験で用いた評価指標は本プロトタイプが有用かどうかを示すための一例として，十分妥当であると考えられる．

5.2 評価結果について

本実験の結果として得られた表 4.1 の結果について詳述する．本実験では git-sql module を使用した実装について，キャッシュ使用の有り無しの場合について処理時間測定を行った．これは SZZ アルゴリズムにおいてはあまり意味がなく，本来ならば SZZ アルゴリズムの処理時間測定の場合，キャッシュ使用なしの場合の計測時間のみで十分である．

本実験でキャッシュの有り無しの場合を計測した理由は，SZZ アルゴリズム以外のリポジトリマイニングのタスクの場合，複数回実行することが前提となるタスクが考えられる．そのようなタスクの場合に於いて，本研究で実装したクエリ言語利用ツールに優位性があるかどうかを考察するためである．

本研究で行った評価では，先述したとおり git-sql module を利用したリポジトリマイニングのタスク実行は従来手法と比べ，実装に必要なコード分量は 2 割未満となり，大幅に削減することが可能であった．

一方，処理時間に関してはキャッシュを用いない場合では従来手法より 7 割多く掛かってしまうという結果となった．これは，従来手法と違い，各々のリポジトリマイニングアルゴリズムとは関係のないデータ（例えばユーザ情報やコミットの diff 情報）まで処理してしまうため，データベースの構築のためにリポジトリへアクセスする時間計算量が従来のリポジトリマイニングアルゴリズムと比べて大きくならざるをえなかったためと考えられる．

しかし，キャッシュを利用できた場合，すなわちデータベースキャッシュの整合性のみをチェックした後に SZZ アルゴリズムを実行した場合，従来手法の 3 割未満の処理時間でタスクを処理することができた．

5.3 評価実験のまとめ

これらの結果より，本研究で実装したプロトタイプによるクエリ言語の活用により，ソフトウェア工学におけるリポジトリマイニングツールの実装における実装コストは非常に低くなると考えられる．

また，移り変わるリポジトリに対して複数回繰り返す必要のあるアルゴリズムでは，キャッシュを活かした高速検索が可能であり，処理時間の点でも十分従来手法を上回りうると考える．

しかし，一回きりの実行が前提となるアルゴリズムで，巨大なリポジトリを対象とするものでは本研究で実装したプロトタイプでは処理時間が非常に大きくなり，場合によっては直接 Git コマンドを用いる場合と比較して大きな差が出てしまう場合が考えられる．この点に関しては本実験でのプロトタイプでは先述したとおり限界があり，今後の課題とする．

上記の通り問題は残るものの，本実験により，クエリ言語をリポジトリマイニングに利用するというアイディアはソフトウェア工学研究者がリポジトリマイニングを行う際のマイニングツールの実装コストを大きく削減できるという結果が得られた．

6. 結言

本研究では，クエリ言語を用いてGitリポジトリへのリポジトリマイニングを行う際の実装コストがどのように変化するかという点について実証した．

本研究でプロトタイプツールを利用して行ったリポジトリマイニングタスクでは実装コスト削減の効果は大きく，クエリ言語を用いてGitリポジトリのマイニングを行うというアイディアは有用であると考える．

また，キャッシュを利用するというアイディアは本質ではないものの，本プロトタイプツールを用いたリポジトリマイニングについて，反復実行するアルゴリズムにおいては処理時間のデメリットも少なく，クエリ言語の活用の一例として十分な実用性があることを示せたと考える．

また，本研究では一度データベースファイルへの変換を行ってからクエリを実行するという方針で実装したプロトタイプについて評価を行ったが，実装方法を変えることで処理時間の高速化や利用しやすさを改善することができると考える．例えば，データベースファイルへの変換を行うのではなく，一旦SQL文をパースしてGitコマンドを構築してやり，それを元にリポジトリへの検索を行う方法が考えられる．この手法では無駄なリポジトリ操作を減少させ，処理時間を短縮できる可能性がある反面，SQL文に含まれる条件式のパースなどが必要であり，本研究でのプロトタイプでは扱うことができなかった．

以上のように本研究で実装したプロトタイプの改良は今後の課題とするが，本研究の目的であったクエリ言語のGitリポジトリマイニングにおける有用性は確認できたと考える．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂いたのみならず，研究環境を整えて頂きました，本学情報工学・人間化学系水野修准教授に厚く御礼申し上げます．

また，本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻河端駿也先輩，山田晃久先輩，采野友紀也先輩，藤原剛史先輩，森啓太先輩，情報工

学課程 黒田翔太君，田中健太郎君，西浦生成君をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します.

参考文献

[1] Git, Git - Fast Version Control System, Git - Fast Version Control System（オンライン）, 入手先〈https://git-scm.com/〉（参照 2016-2-8）.

[2] 都司達夫，宝珍輝尚，データベース技術教科書—DBMS の原理・設計・チューニング，CQ 出版社，東京，2003.

[3] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?,” ACM sigsoft software engineering notes, vol.30, no.4, pp.1–5, ACM, 2005.

[4] G. Gousios and D. Spinellis, “Ghtorrent: Github’s data from a firehose,” Mining software repositories (msr), 2012 9th ieee working conference on, pp.12–21, IEEE, 2012.

[5] sqlite.org, SQLite Home Page, sqlite.org（オンライン）, 入手先〈https://www.sqlite.org/〉（参照 2016-2-14）.

[6] Python Software Foundation, Python 2.7 Release, Python.org（オンライン）, 入手先〈https://www.python.org/download/releases/2.7/〉（参照 2016-2-14）.

[7] Michael Trier and contributors, GitPython 1.0.2 documentation, GitPython Documentation（オンライン）, 入手先〈https://gitpython.readthedocs.org/〉（参照 2016-2-14）.

[8] The Eclipse Foundation, BIRT Home, BIRT Home（オンライン）, 入手先〈http://www.eclipse.org/birt/〉（参照 2016-2-8）.

[9] eclipse.org, eclipse/birt: The open source Eclipse BIRT reporting and data visualization project, GitHub（オンライン）, 入手先〈https://github.com/eclipse/birt〉（参照 2016-2-14）.

[10] eclipse.org, Simply Search, bugs.eclipse.org（オンライン）, 入手先〈https://bugs.eclipse.org/bugs/query.cgi?product=BIRT〉（参照 2016-2-14）.

[11] O. Mizuno, szz_tool, o-mizuno / szz_tool - Bitbucket（オンライン）, 入手先〈https://bitbucket.org/o-mizuno/szz_tool〉（参照 2016-2-12）.

[12] Perl.org, The Perl Programming Language, www.perl.org（オンライン）, 入手先〈https://www.perl.org/〉（参照 2016-2-14）.

[13] RedMonk, Programming languages ranked by expressiveness, RedMonk（オンライン）, 入手先〈http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/〉（参照 2016-2-8）.