Incremental Map-Reduce on Repository History

Johannes Härtel
University of Koblenz-Landau
Software Languages Team
Germany
johanneshaertel@uni-koblenz.de

Ralf Lämmel
University of Koblenz-Landau
Software Languages Team
Germany
laemmel@uni-koblenz.de

***Abstract*—Work on Mining Software Repositories typically involves processing abstractions of resources on individual revisions. A corresponding processing of abstractions of resource changes often depends on working with all revisions of the repository history to guarantee a high resolution of the measured changes. Abstractions of resources and abstractions of resource changes are often very related up to the point that they can be used interchangeably in the processing. In practice, approaches working with abstractions processed over high revision counts face a scalability challenge. In this work, we contribute to the challenge by incrementalizing the processing of repository resources and the corresponding abstractions. Our work is inspired by incrementalization theory including insights on Abelian groups, group homomorphisms and indexing. We provide a map-reduce interface that enables calls to foreign functionality and convenient operations for processing abstractions, such as mapping, filtering, group-wise aggregation and joining. Apache Spark is used for distribution. We compare the scalability of our approach with available MSR approaches, i.e., with LISA that reduces redundancy and with DJ-Rex that migrates an analysis to a distributed map-reduce framework.**

***Index Terms*—mining software repositories, functional programming, repository history, abelian groups, incrementality, distribution, map-reduce.**

I. Introduction

Work on Mining Software Repositories typically involves processing abstractions of resources on single revisions, e.g., source code lines [1], [2], [3], [4], [5], file extensions [6], abstract syntax trees [7], documentation [8], architecture [9], similarity [10], [11], [5], [12], [13], [14] and dependencies [15], [16], [17], [18]. A corresponding processing of abstractions of resource changes, such as the added or removed source code lines, file extensions or AST nodes, often depends on working with all revisions of the repository history to guarantee a high resolution of the measured changes.

Abstractions of resources and abstractions of resource changes are critical for several data analysis tasks. Abstractions mined from a repository, such as source code metrics and their corresponding changes, are important predictors for future changes and bugs [19], [20], [21], [22], [23], [24], [25], [26]. Past changes of code increase the likelihood of a pull request being accepted [27]. Moreover, changes are deeply related to approaches profiling developers [28], [29]. The extraction of abstractions of resources and abstractions of resource changes are essential parts in most empirical studies mining software repositories.

A. The Scalability Challenge

In practice, approaches relying on abstractions processed over high revision counts face a scalability challenge.

A simple way to achieve scalability is to reduce the revision count by sampling. However, this may cause problems in accuracy as changes between sampled revisions are lost.

The scalability challenge can also be handled in terms of parallel and distributed analysis [30], [31], [32], [33], [34]. Such solutions rely on code written in map-reduce [30], [31] or domain specific languages [32], [33], [34]. However, they scale at the expense of high hardware usage.

Hence, some approaches report on ad hoc incrementalization strategies to accelerate the processing [35], [36]. The typical analysis code calls *git-diff* and just examines the changed resources between revisions. Such incremental analysis may be complicated, because it requires several additional bookkeeping efforts, not applicable for prototyping [30].

A related solution is to reduce redundancies leveraging the high similarity between succeeding revisions. One approach exposing this mechanism is described in [37], [38], [7] which requires analysis to be written in Signal/Collect [39].

B. The Topleet Solution

Abstractions of resources and abstractions of resource changes are often very related. We show how to use them interchangeably in the incremental analysis of repository history, based on theoretic insight on Abelian groups, group homomorphisms and indexing [40], [41].

In particular, our discussion includes incrementalization strategies that can be used for incremental mapping, filtering, group-wise aggregation and joining of resources and the corresponding abstractions. Our description can be viewed as an instruction set for the efficient realization of MSR studies. Further, we introduce a prototype called Topleet[1], providing a map-reduce interface for the incremental and distributed processing of repository history. Topleet analysis is embedded into regular Scala code and provides a syntax that resembles non-optimized map-reduce code, widely understood and used for MSR [30], [31]. Our approach introduces moderate constraints on the involved types used in the map-reduce calls.

[1]**Implementation, advanced examples, usage guidelines and supplementing evaluation data** are available under http://github.com/topleet/topleet.

We compare the scalability of Topleet to an approach relying on a manual migration of code to distributed map-reduce [30], [31] and to a reduction of redundancies [37], [38], [7].

C. Summary of the Paper's Contributions

- Topleet; a scalable MSR approach for processing the repository history in map-reduce style.
- An evaluation comparing the performance of Topleet with LISA and DJ-Rex and an evaluation of Topleet's infrastructure.
- An adaptation of incrementalization theory able to explain common habits in processing repository history.

D. Road-map of this Paper

Sec. II discusses different solutions to processing repository history from an application-oriented perspective. Sec. III describes the basics of our approach including how to use Abelian groups to represent changes. Sec. IV describes how to exclusively process a collection of changes while preserving the intuitive semantics of map-reduce. Sec. V describes in which cases this is not possible and introduces index based processing. Sec. VI focuses on related infrastructure concerns. Sec. VII presents the evaluation of the prototype. Sec. VIII discusses related work. Sec. IX concludes the paper.

II. Hands on Processing Repository History

We start with presenting an application-oriented view on recent solutions to processing repository history. As the application task, we choose to compute McCabe's cyclomatic complexity on all Java file revisions in a repository. The computation of cyclomatic complexity has been used for the presentation of LISA [37], [38], [7], it fits into the Boa infrastructure [32], [33], [34] and can be migrated to a distributed map-reduces frameworks in analogy to DJ-Rex [30], [31].

A. Migration to Distributed Map-Reduce (DJ-Rex)

DJ-Rex is presented in [30], [31] as a proof-of-concept showing the migration of an existing MSR analysis to distributed map-reduce. We reproduce two solutions, according to the presentation of DJ-Rex, migrating a function that computes McCabe's cyclomatic complexity to a distributed map-reduce framework. We call the solutions 'DJ-Rex' and 'DJ-Rex Optimized'. The former includes the following code:

```
1 val resources: RDD[(SHA, Path, Resource)] = . . .
2 val mcCabe: RDD[(SHA, Int)] = resources
3     .filter { case (_, path, _) => path.endsWith(".java") }
4     .map { case (sha, path, resource) =>
5         (sha, computeMCC(resource))}
```

Listing 1. DJ-Rex Solution: An excerpt of our reproduction of [30] migrating `computeMCC` to distributed map-reduce.

In particular, we use Scala and Apache Spark for distribution. The Resilient Distributed Datasets (RDDs) [42] provide a distributed collection of elements that can be operated on in parallel. The operation `filter` restricts the resources on Java files and `map` calls the foreign function `computeMCC` to compute the cyclomatic complexity on resources. We thereby executed the processing in parallel and distributed manner.

The problem with such simple code is that, already for a medium-sized repository, like libgdx/libgdx[2], with around 14.000 commits, it needs approximately seven hours on a single machine. Manual incrementalization efforts optimizing this map-reduce code to just handle changed resources ('DJ-Rex Optimized') leads to a much faster solution taking six minutes. Accomplishing the same task in the same time, without change processing but just adding hardware, would require 70 machines instead of one.

B. Domain Specific Languages (Boa)

Another approach that can be used to analyze repository history on a distributed map-reduce platform is Boa [32], [33], [34]. To this end, the analysis is written in a domain specific language (DSL). The following excerpt can be found in the reference documentation of BOA[3] answering the question: 'How many fixing revisions added null checks?'. We show the original solution; an adaptation that computes the cyclomatic complexity is straightforward and omitted here.

```
1  before node: ChangedFile -> {
2      // if this is a fixing revision and there was a previous version of
           the file
3      if (isfixing && haskey(files, node.name)) {
4          // count how many null checks were previously in the file
5          count = 0;
6          visit(getast(files[node.name]));
7          last := count;
8          // count how many null checks are currently in the file
9          count = 0;
10         visit(getast(node));
11         // if there are more null checks, output
12         if (count > last)
13             AddedNullCheck << 1;
14     }
15     if (node.change == ChangeKind.DELETED)
16         remove(files, node.name);
17     else
18         files[node.name] = node;
19     stop;
```

Listing 2. Boa Solution: 'How many fixing revisions added null checks?' (Adapted copy from the Boa reference documentation)

To determine if null checks have been added, the code counts null checks on the previous revision of a changed file (if there is one) and on the current version. Comparable to the DJ-Rex Optimized solution (omitted in this paper), Boa code addresses changes explicitly – eye-catching by the usage of types such as `ChangedFile` and `ChangeKind`.

Boa provides a web-based interface to the proprietary Boa infrastructure [33]. We do not report on its performance as we cannot reproduce this setup. However, we assume that the performance does not significantly differ from the DJ-Rex solutions; we also assume that there is the same performance gap between change-oriented and non-change-oriented treatment.

[2] http://github.com/libgdx/libgdx

[3] http://boa.cs.iastate.edu/docs/index.php

C. Reduction of Redundancies (LISA)

LISA [37], [38], [7] is a solution that reduces the redundancies in the multi-revision code analysis and the first in this presentation that does not require the explicit handling of changes. LISA needs a registered parser for a target file extension and analysis code written in Signal/Collect [39]. The code for the computation of McCabe's cyclomatic complexity can be found in the publications [38], [7]. LISA does not allow direct calls to foreign functions.

The computation of McCabe's cyclomatic complexity on the same repository requires 12 minutes; however, LISA has the highest memory footprint with 6.4 GB.

III. Topleet

The following code is integrated into standard Scala. It shows Topleet computing the cyclomatic complexity on all Java file revisions in the repository libgdx/libgdx.

```
1  // Git initialization.
2  val shas: Leet[SHA] = git("libgdx/libgdx")
3  // Accessing the resources of each commit.
4  val resources: Leet[Bag[(Path, Resource)]] = shas.resources()
5  // Filtering for Java.
6  val javas: Leet[Bag[(Path, Resource)]] = resources
7      .filter { case (path, resource) => path.endsWith(".java") }
8  // Computing the metrics.
9  val mcCabe: Leet[Bag[Int]] = javas
10     .map { case (path, resource) => computeMCC(resource)}
11 // Summing up the metrics.
12 val summed: Leet[Int] = mcCabe.sum()
```

Listing 3. Topleet Cyclomatic Complexity Solution: The part from line 4 to 10 corresponds to the map-reduce steps of the DJ-Rex solution, shown in Listing 1.

The code uses the Topleet data structure `Leet[V]` which maintains the repository history as a graph in the background. Commits (revisions or version) are nodes and corresponding relationships (e.g., committing or forking) are edges. The data structure populates each node of this background graph with one data entry of the generic type `V`.

Method calls on the data structure invoke processing steps that return new data structures with altered data entries. This enables a fluent API syntax that sequentially applies processing steps to the data entries of all commits, similar to map-reduce. For clarity, we assign the intermediate results in Listing 3 to placeholders (`val`) annotated by the type.

The solution starts with a call to `git`, initializing the first data structure with the commit history of repository libgdx/libgdx used as background graph. It populates each node of the background graph (i.e., a commit) with a data entry of type `SHA` reflecting the commit's identity. Method `resources()` is invoked on this data structure to read out the available path-resource tuples for each `SHA`, i.e., the commit's corresponding resources at this point in time. The returned data structure maintains data entries of type `Bag[(Path, Resource)]`. The type `Bag[E]` is a bag of elements of type `E`. `Path` is a path to a resource and `Resource` is a pointer to repository content providing an input stream. The next steps invoke `map`, `filter` and `sum` according to the standard semantics of bags; `computeMCC` is a foreign function of type `Resource` $\rightarrow$ `Int` to compute the cyclomatic complexity on a single resource; it is the parameter of the function `map`.

Topleet processes the libgdx/libgdx history in around 3.2 minutes (DJ-Rex Optimized takes 6.5 minutes, LISA 12.0 and DJ-Rex 449 minutes). Topleet includes the change processing under the hood but looks almost similar to the DJ-Rex solution.

A. Comparison with the DJ-Rex Solution

The code highlighted in the mid of Listing 3 directly corresponds to the DJ-Rex solution shown in Listing 1. DJ-Rex uses the distributed collection type `RDD[V]` for the processing of data entries. While the collection elements, such as `(SHA, Path, Resource)`, explicitly refer to the commits by containing a `SHA`, Topleet uses the custom type `Leet[V]`, associating a commit with a data entry `V` internally. The map-reduce invocation `filter` and `map` are used in both solutions. However, instead of associating each path-resource pair with a commit, Topleet associates a bag of path-resource pairs `Bag[(Path, Resource)]` with each commit.

B. Core Operations

Topleet composes any functionality, such as the methods used in the solution, by five core operations. We will shortly revise their implications on the overall approach and refer to the sections where they are discussed.

- Topleet uses Abelian groups to represent changes of the data entries along the repository history. The core operations `zero`, `merge` and `inv` correspond to the group operators of Abelian groups and are discussed in the remainder of this section.
- Sec. IV will explain the *change processing*. The core operator `tmapHom` is a foreign function interface that permits processing changes; sufficient to realize basic map-reduce functionality, such as `map`, `filter`, `sum` and `count`. It is limited to just invoking functions that conform group homomorphisms.
- Sec. V will explain the *index processing*. The processing of changes is not appropriate for every foreign function call, e.g, `join`, `distinct` and group-wise aggregation can not be realized. The core operation `tmapIx` is a foreign function interface that enables the optimized implementation of such operations by using an index.

The core operations can also be used to plug MSR specific extensions, dedicated for the analysis of Git repositories while decoupled from Topleet's core. Sec. IV includes a discussion on how *git-diff* is plugged using the core operation `tmapHom`.

C. Background Graph

The Topleet data structure `Leet[V]` hides the repository history under the hood in terms of the *background graph*. The background graph is created during the initialization of the data structure and remains unchanged from this point on. It consists of i) nodes `N` that represent individual states in the history, such as commits, revisions or versions, and of ii) directed edges `(N, N)` connecting very similar states, by relations, such as committing, forking and updating.

The background graph is not limited to repository history and may also reflect other evolving artifacts, for instance, the versions of a JAR file and the respective semantic versioning relation. Versioning practice, like the usage of branches, may affect the topology of the background graph being a sequence, tree or acyclic graph.

D. Change Collection

An *index* assignment of a node `N` refers to the data entry `V` assigned to the corresponding commit.

Instead of maintaining this index `V` at every node `N` by a collection, such as `Map[(N, V)]`, Topleet maintains a collection of index changes in terms of `Map[(N, N), V]`. Each element in such a *change collection* consists of an edge `(N, N)` defining where in the background graph a change `V` alters the index assignment. To this end, the index and its change need to be used interchangeably, being of the same type `V`.

We illustrate the correspondence between index and change in `V` by natural numbers for `Leet[Int]`. While the index assignment is a number, an addition can be expressed as a positive number, a removal as a negative number and no change by `0`. If node `N1` is assigned to index `5` and node `N2` is assigned to index `7` (denoted by $ix\ (N1) = 5$ and $ix\ (N2) = 7$), we can assign an edge `(N1, N2)` to the change $\Delta\ (N1, N2) = ix\ (N2) - ix\ (N1) = 2$. We can add the element `((N1, N2), 2)` to the change collection maintained by `Leet[Int]`.

E. Abelian Groups

Clearly, there is no immediate benefit in maintaining a natural number or its change. But, the key contribution becomes clearer when generalizing this treatment to Abelian groups.

> **Abelian Group**
> An Abelian group is implemented by a merge operator $*$ that combines two elements in V returning an element in V, an inverse operator e^{-1} in V, and the zero element z in V. The operator $*$ is commutative.

The Abelian group for natural numbers `Int` defines addition to be the merge operator $*$, negation to be the inverse e^{-1} and 0 to be the zero element z.

In general, an Abelian group corresponding to type `V` of `Leet[V]` is used to derive the underlying change collection. For each background graph edge `(N1, N2)` and the corresponding index assignments $ix\ (N1)$ and $ix\ (N2)$, a change element `((N1, N2),` $ix\ (N1)^{-1} * ix\ (N2)$`)` is created.

In order to derive the change collection for `Leet[Int]` we can use the Abelian group for `Int`. However, for covering all types of the solution, we still miss an Abelian group for `SHA`, `Bag[(Path, Resource)]` and `Bag[Int]`.

F. Composing Abelian Groups

One of the fundamental types that is used to provide the missing Abelian groups for the cyclomatic complexity solution is `Map[K, V']` which has a corresponding Abelian group if there is a 'nested' Abelian group for `V'`. The Abelian group operators for type `Map[K, V']` are defined as follows: The operator e^{-1} inverts the values of the map by the nested e^{-1} operator; the $*$ operator merges two maps in that it combines the values with the same key by the nested $*$ operator filtering out nested z elements; operator z returns an empty map.

Accordingly, we can use the Abelian group for `Int` to compose an Abelian group for `Map[K, Int]`. For instance, computing the change between two nodes with the index assignment `{(A→1)}` and `{(B→1)}` gives us the change `{(A→-1),(B→1)}`, assigning `A` to `-1` and `B` to `1`. The type `Map[K, Int]` can also be considered as an alias for `Bag[K]` assigning bag element `K` to the number of its occurrence. We use this Abelian group for the data structure `Leet[Bag[(Path, Resource)]]` and `Leet[Bag[Int]]` in our cyclomatic complexity solution.

Other than that, we have a corresponding Abelian group for tuples `(V1,V2)` that can be composed out of an Abelian group for `V1` and `V2` by delegating the group operators to the nested groups.

G. Escaping Abelian Groups

The processing of a Git repository, as shown in the solution, usually starts with the initial data structure `Leet[SHA]`. However, the type `SHA` misses a corresponding Abelian group and can thereby not directly be represented in a change collection.

In principle, we can wrap an arbitrary type `T`, missing an Abelian group, by `Bag[T]` and get an Abelian group that describes the change without further decomposition of `T`. In the case of `SHA`, this is `Map[SHA, Int]` (alias `Bag[SHA]`) pointing from a removed `SHA` to `-1` and from an added `SHA` to `1`. But, finding a reasonable decomposition of `T` into parts that change little along the edges of the background graph remains important for the efficient processing by just handling changed parts.

While the repository content is predestined to be decomposed into bags of resources that change little (i.e., by `Bag[(Path, Resource)]`), decomposing type `SHA` is not beneficial. Neither would a decomposition lead to small changes, as succeeding SHAs are not trivially related, nor would it enable meaningful operations on changing parts. By default, we substitute a type `T` by `Bag[T]` if there is no Abelian group for `T`.[4]

H. Abelian Core Operations

Abelian groups provide us with Topleet's core operations `merge`, `inv` and `zero`, defined according to an Abelian group for the change collection `Map[(N, N), V]`. The binary `merge` operator combines two change collections of the same type `V`; `zero` creates an empty change collection; the `inv` operation inverts the changes `V`.

[4]In the Scala solution code, Abelian groups are listed in a library and inferred by type during compilation using implicit parameters. Realization of Topleet without implicit parameters is not as convenient.

IV. Change Processing with Topleet

So far, we have a background graph reflecting the history of a repository, nodes reflecting commits with assigned data entries, and a collection of changes of data entries derived by a corresponding Abelian group. We now revise the computation steps of the cyclomatic complexity working only on the change collection. A detailed sample of such processing is given in Fig. 1.

A. Initialization

The processing starts with a call to `git` taking the repository address as parameter. It initializes the data structure `Leet[SHA]` by creating the background graph and populating the corresponding change collection. The Abelian group used corresponds to type `Bag[SHA]` which is an alias for `Map[SHA, Int]` and the internal substitute for the escaped type `SHA`.

We exemplify such initialization on a simple background graph that consist of three succeeding commits A, B and C. We use letters to identify commits instead of real SHAs. To derive the first change collection, we first define the index assignment of the background graph nodes: $ix\ (A) =$ `{(A→1)}`, $ix\ (B) =$ `{(B→1)}` and $ix\ (C) =$ `{(C→1)}`. We also use a short-hand notation for bags, i.e., `{A}`, `{B}` and `{C}`. The first change collection lists the changes: $\Delta\ (A, B) =$ `{(A→-1),(B→1)}` removing `A` and adding `B` (short `{`$\overline{\text{A}}$`,B}`) and $\Delta\ (B, C) =$`{(B→-1),(C→1)}` removing `B` and adding `C` (short `{`$\overline{\text{B}}$`,C}`). From now on, the processing commences on the change collection with the elements `((A, B), {`$\overline{\text{A}}$`,B})` and `((B, C), {`$\overline{\text{B}}$`,C})`.

B. Group Homomorphism

For the remaining steps, the way in favor is to process the elements of the change collection. Intuitively, applying a function h to an element in the change collection does not correspond to applying it to the original indexes from which the changes have been derived. However, for all processing steps used in the computation of the cyclomatic complexity, the indexes and the changes can be processed interchangeably by the same functions, without altering the semantics.

In general, the change collection can be processed if the applied function $h : G \to H$ conforms a group homomorphism between two Abelian groups used for the input change collection G and output change collection H.

> **Group Homomorphism**
> Given two groups $(G, *)$ and $(H, \cdot)$ with the group operators $*$ and $\cdot$, a group homomorphism h is a function between G and H for that $h(u) \cdot h(v) = h(u * v)$ holds.

Consider two connected commits A and B with the index assignments u and v, the derived change $u^{-1} * v$ and the corresponding change element `((A, B),`$u^{-1} * v$`)`. Fig. 2 shows the application of a group homomorphism h where dotted lines depict the application of h, and arrows the edges in the background graph before and after the application.

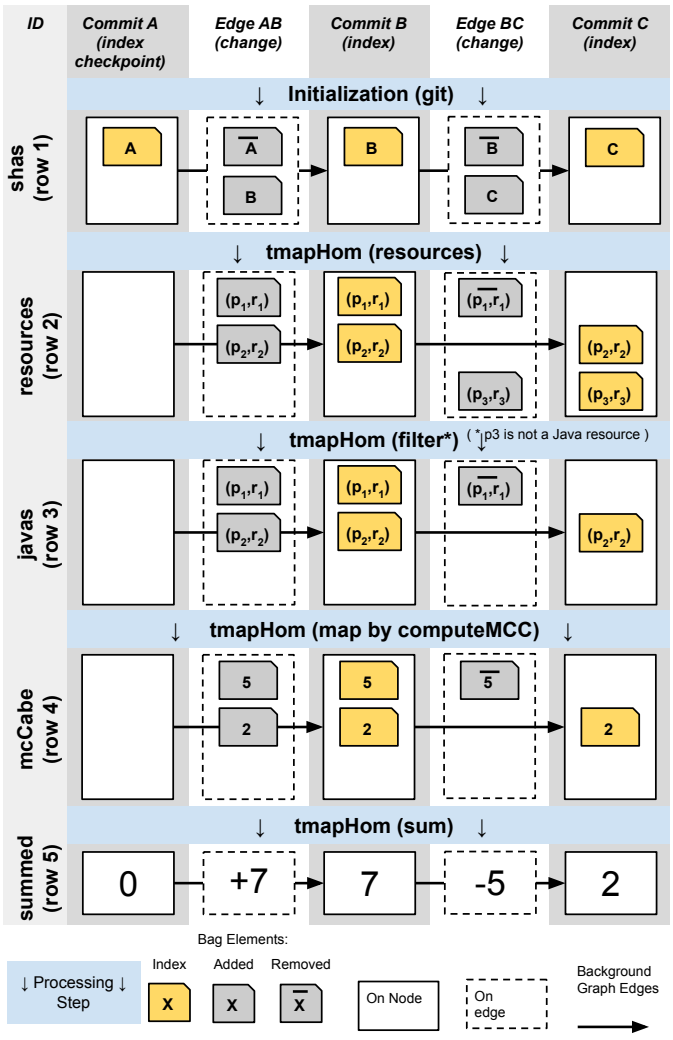


Fig. 1. **Interchangeable processing of abstractions of resources and abstractions of resource changes:** The plot depicts the processing steps (top to bottom) computing the cyclomatic complexity on three succeeding commits (left to right). **Row 1**: The indexes and changes are initialized according to an Abelian group for `Bag[SHA]`. **Row 1** $\to$ **2**: The path-resource tuples are extracted. Commit A is empty, commit B adds the tuples (p_1, r_1) and (p_2, r_2); commit C removes tuple (p_1, r_1) and adds (p_3, r_3). **Row 2** $\to$ **3**: We assume that path p_3 points to a Bitmap; hence, the tuple (p_3, r_3) is filtered out. **Row 3** $\to$ **4**: The metrics are computed given by type `Bag[Int]`. Resource r_1 has a cyclomatic complexity of `5` and r_2 a cyclomatic complexity of `2`. **Row 4** $\to$ **5**: Finally, the metrics are summed up. This is a group homomorphism between bags of natural numbers `Bag[Int]` and natural numbers `Int`. The change at edge AB becomes `7` and the change at edge BC `-5`.

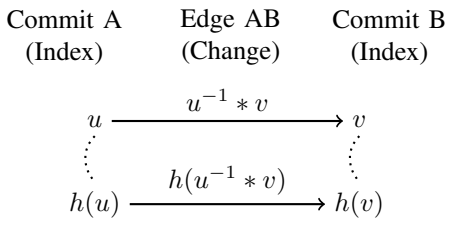


Fig. 2. Application of a group homomorphism h on the index assignments u and v and the corresponding change on edge AB $u^{-1} * v$ for that $h(u)^{-1} \cdot h(v) = h(u^{-1} * v)$ holds.

The properties of the group homomorphism h assure that the new change, produced by the function application $h(u^{-1} * v)$ on the previous change $u^{-1} * v$, is the same as if it is computed on the indexes by $h(u)^{-1} \cdot h(v)$. Hence, we can simply map the previous change collection element to the current change collection element `((A, B),h(`$u^{-1} * v$`))`.

Since we define the edges in the background graph to connect very similar states in the history, we expect the average changes to be much smaller than the indexes. Exclusively applying an expensive function h to the much smaller change is one central aspect of the incrementalization.

The core operation `tmapHom` provides the interface for this functionality. It applies a function $h : V1 \rightarrow V2$ to the elements of the change collection of `Leet[V1]` and returns the new change collection in terms of `Leet[V2]`. The background graph is preserved during such invocation.

C. Git-Diff

Currently, the change collection reflects the changing commits by the elements `((A, B), {`$\overline{\text{A}}$`, B})` and `((B, C), {`$\overline{\text{B}}$`, C})`. To continue with the computation of the cyclomatic complexity, we need to extract the available path-resource pairs for every commit – or even better, we need to extract the changed path-resource pairs between commits.

A prominent function that does so is *git-diff*. To have corresponding Abelian groups, we define the input of *git-diff* to be of type `Bag[SHA]` and the output to be of type `Bag[(Path, Resource)]`[5]. The input of *git-diff* is an alias for `Map[SHA, Int]` such that the previous commit can be assigned to -1 (negative bag element) and the next commit to 1 (positive bag element). The output is an alias for `Map[(Path, Resource), Int]` such that removed path-resource pairs can be assigned to -1 (negative bag element) and added pairs to 1 (positive bag element).

Accordingly, we can apply *git-diff* to a single commit, e.g., to the bag `{B}`, extracting the bag of all path-resource pairs available at this commits; or we can apply *git-diff* on the change between two commits, e.g., `{`$\overline{\text{B}}$`,C}`, extracting the difference between both commits. For instance, a difference could look like `{`$\overline{(\text{p}_1,\text{r}_1)}$`,(`$\text{p}_3$`,`$\text{r}_3$`)}` removing `(`$\text{p}_1$`,`$\text{r}_1$`)` and adding `(`$\text{p}_3$`,`$\text{r}_3$`)`.

This definition of *git-diff* meets the requirements of a group homomorphism; hence, we can use the core function `tmapHom` to apply *git-diff* to the change collection. We provide the method `resources()` that realizes this functionality for Git repositories on the data structure.

D. Basic Map-Reduce Operations

The next steps in the computation of the cyclomatic complexity invoke `map`, `filter` and `sum` on the changing resources. Such operations are, as well, group homomorphisms, implemented as foreign functions on top of the core method `tmapHom`. The following map-reduce operations work according to the standard semantics of bags and maps and can be applied to changes and indexes interchangeably:

- The implementation of `map` applies a function to the keys of `Map[K, V']`. As we use `Map[(Path, Resource), Int]` to represent repository resources, we can use `map` to invoke foreign functions, like `computeMCC`, on the path-resource pairs.
- The implementation of `filter` filters the keys of `Map[K, V']`.
- The implementation of `sum` is applied to `Map[Int, Int]` (alias for `Bag[Int]`). Each element contributes to the sum multiplying the key by the value before aggregation.
- The implementation of `count` sums up the values of `Map[K, Int]` (alias for `Bag[K]`).

V. Index Processing with Topleet

Apparently, applications may require the cyclomatic complexity metrics at commits instead of the metric changes between commits. To this end, a commit's corresponding data entry needs to remain accessible during the processing steps. Moreover, there are extended map-reduce operations that require steps beyond plain change processing. We introduce *index processing* by our last core method `tmapIx`.

A. Index Traversal

Our data structure maintains changes of data entries assigned to commits, but it does not provide direct access to such data entries. Hence, a data entry assigned to a commit (i.e., a node's index assignment) needs to be restored.

Having the change $\Delta\ (N1, N2)$ between two nodes while knowing the index assignment of one node $ix\ (N1)$, the index of node N2 can be recomputed merging the index and the change by the corresponding group operator $ix\ (N2) = ix\ (N1) * \Delta\ (N1, N2)$. More generally, changes can be merged along a path in the background graph in that the index of nodes reachable over some known index can be recomputed. In an acyclic background graph, different paths are interchangeable as the change relation satisfies the triangular equation $\Delta\ (N1, N2) * \Delta\ (N2, N3) = \Delta\ (N1, N3)$.

An *index traversal* starts on a node with a known index assignment and traverses the edges of the background graph. During such traversal, this index is successively updated by merging the traversed edge's change contained in the change collection. The traversed index assignments can be listened. Immutable data types may help to avoid visiting nodes twice or copying the index when the history branches.

B. Checkpoint Collection

A problem of an index traversal is that it needs at least one index assignment to start with. Without such reference point, the relative change collection can not be transposed to the original indexes it has been derived from. To enroll an index traversal, the Topleet data structure maintains a second *checkpoint collection* storing the index assignment of one node

[5]We extract added and removed resources while considering updated resources as adds and removes at the same path. A fine-grained decomposition of the output, e.g., into added and removed lines, is possible by adjusting the Abelian group for the output.

in every connected component of the background graph (e.g., by `Map[N, V]`). The checkpoint and change collection are processed in analogy by the core operations.

C. Index Application

The *index application* applies a function $f : V1 \to V2$ to the index assignments restored during an index traversal of `Leet[V1]` and derives the changes of a new data structure `Leet[V2]` on the fly. Such application is inevitable if the function f does not conform a group homomorphism. The interchangeable application of f to changes and indexes would alter the semantics as $f(u)^{-1} \cdot f(v) = f(u^{-1} * v)$ does not hold in general.

D. Key Index Application

Applying a function f repeatedly to huge indexes can be cost intensive. To avoid this, a second form of the index application is introduced that just applies f to parts of the index that change. We use the function $f : (K1, V1) \to V2$ for the interface of `tmapIx` applied to `Leet[Map[K1,V1]]` with the output `Leet[V2]`.

Consider the following case where key k_1 can be canceled out as it does not change. We have two connected commits A and B with the index values $\{(k_1 \to w), (k_2 \to u)\}$ and $\{(k_1 \to w), (k_2 \to v)\}$. We have the change collection element `((A,B),`$\{(k_2 \to u^{-1} * v\})$`)`.

The regular index application traverses A and B, restores the index assignments and derives the new change element after applying f, i.e., `((A, B),`$f(k_1, w)^{-1} \cdot f(k_2, u)^{-1} \cdot f(k_1, w) \cdot f(k_2, v)$`)`. However, the application of f to the unchanged key k_1 can be canceled out in this result. The new change element in the collection is just `((A, B),`$f(k_2, u)^{-1} * f(k_2, v)$`)`. Limiting the application of f to changed keys during the traversal is the second incrementalization strategy in our approach used in `tmapIx`.

E. Extended Map-Reduce Operations

Extended map-reduce functionalities, like `distinct`, `join`, `cogroup`, `cartesian` and different group-wise aggregation schemes, rely on `tmapIx` and index traversing.

VI. Computation Infrastructure

Topleet can be realized on local or distributed computation infrastructures that provide map-reduce functionality on collections. In this section, we discuss how to adapt the five core operations to efficient processing infrastructures. It is based on our experience implementing Topleet on Scala Collections and Apache Spark's Resilient Distributed Datasets (RDDs).

A. Lineage

Lineage refers to operations being chained and invoked on collections without materializing intermediate results [43], [44]. Topleet delegates the realization of data lineage to the background infrastructure. To guarantee lineage by an element-wise processing of the change and checkpoint collections, we adapt the original collection definition, e.g., `Map[(N, N), V]`, to plain sequences `Seq[((N, N), V)]`. It allows that changes on edges split into multiple collection elements. For instance, `((A, B),` $u * v$`)` can be split into the collection elements `((A, B),` u`)` and `((A, B),` v`)`. This makes no semantic difference for the application of a group homomorphism as $h(u*v) = h(u) \cdot h(v)$ holds. The operations `tmapHom`, `merge` and `inv` can be chained without materializing and merging intermediate results.

Merging corresponding changes and checkpoints may decrease the size of the collections. Deciding if changes should be merged depends on the function and the data. We set corresponding defaults for the map-reduce functions and defer additional optimization to future work.

B. Distribution

We delegate the distribution to Spark's Resilient Distributed Datasets (RDDs) [42]. To enable the fine-grained partitioning of large changes and checkpoints, we set `V` in `Leet[V]` to `Map[K, V']` by default. The corresponding RDDs maintain key-value pairs for changes `RDD[((N, N), (K, V'))]` and checkpoints `RDD[(N, (K, V'))]`. We revise the interface for `tmapHom` and `tmapIx`, defined now as $f : (K1, V1) \to Map[K2, V2]$. The processing of arbitrary Abelian groups is possible by wrapping `V` in `Map[K, V]`.

The core operations `tmapHom` and `inv` do an element-wise mapping of such distributed collections and `merge` appends the respective change and checkpoint collections. For `tmapIx`, which needs to enroll an index traversal, we use a two-dimensional coordinate to shuffle collection elements.

- **Width:** The width coordinate refers to a partitioning by hash of the element's key `K`.
- **Height:** The height coordinate refers to a partition of the background graph in which the element's edge `(N,N)` or node `N` is located. A partitioning of the background graph into connected components is preferred as every component needs a checkpoint.

To apply `tmapIx`, the change and checkpoint collection elements are shuffled in that those with the same width and height are moved to the same partition. Multiple assignments of `K` to `V` can be merged during the shuffle step according to the respective Abelian group for `V`. Afterwards, `tmapIx` can be applied within a partition containing all necessary key changes and checkpoints for an index traversal of the background graph maintained in this partition.

C. Memoization

Applying a cost-intensive function (such as parsing) on the same input twice can be circumvented by caching a function's corresponding input and output pairs or by applying the function on an inverted representation of the collections (e.g., applying `f` to the keys of `RDD[(K, Set[(V, N, N)])]`). Memoization is another open parameter that depends on the particular function and data.

VII. Evaluation

We evaluate Topleet in computing McCabe's cyclomatic complexity for all Java file revisions in a repository because

it fits all compared approaches and is originally used for the presentation of LISA. The online resources contain additional tasks and supplementing evaluation results.

A. Solutions

We compare Topleet in different configurations with two manual migrations DJ-Rex and DJ-Rex Optimized (presented in Sec. II and available online); and with LISA (the solution code is presented in the publications [38], [7]). We cannot compare to Boa (presented in [32], [33], [34]) because we cannot reproduce the proprietary infrastructure behind it.

B. Software, Hardware and Default Parameters

All solutions are configured to the best of our knowledge. We follow a list of principles to assure that the evaluation is as objective as possible:

1) We use the same Java parser. 2) For DJ-Rex (Optimized) and Topleet we used Apache Spark for distribution. If running Spark on a single machine, we fully employ the capabilities by using the local mode with 16 cores. We use Kryo serialization with a buffer size of $512m$. 3) For Topleet and DJ-Rex (Optimized), we use one partition for each 100 commits with a minimum of 32 partitions to guarantee parallelism. We assigned an equal amount of partitions to Topleet's width and hight. 4) We patched the `tick-duration` of Apache Akka to 10 milliseconds to run LISA on Windows. 5) Solutions that depend on `computeMCC` use the same implementation. 6) We exclude the summation of the cyclomatic complexity to align the output granularity of all approaches on the file level. 7) The output is fully persisted to a single storage system. We used the `collect` mechanism of Apache Spark and LISA's default CSV persistence. 8) The output of the approaches differs: LISA persists metric values for linear ranges of the *flattened commit history*[6], DJ-Rex (Optimized) persists metric changes for the flattened commit history and Topleet changes for the commits of the acyclic commit history. 9) Depending on the distribution mode, all solutions are executed on the same hardware. The local evaluation is executed on an Intel Core i5-6600 @ 3.30GHz with 32GB memory, 64-bit, Windows 10. We isolate each run in a separate JVM. Distribution is evaluated using 4 or 7 Amazon EMR m5.xlarge on demand instance with 4 virtual cores and 16GB memory each. 10) Local measurements exclude the time for downloading a repository.

C. Variability

For Topleet, we explore the following configurations:

- **Topleet**: We refer to the solution performing best as Topleet. It uses Apache Spark, `tmapHom` for the `map` including memoization and `filter` preserving lineage.
- **Topleet (Mem. Off)**: Topleet without memoization.
- **Topleet Scala Collections (Mem. Off)**: The Scala Collections implementation, no memoization.
- **Topleet Index Based (Mem. Off)**: Topleet without memoization and `map` using `tmapIx`.
- **Topleet 4x m5.xlarge**: Topleet on one master and 3 cores.
- **Topleet 7x m5.xlarge**: Topleet on one master and 6 cores.

D. Repository Sample

We execute the evaluation on a sample of 98 repositories based on the GHTorrent data set [45] from 2019-06-01[7] which is a mirror of the data exposed by the GitHub API. We sampled for Java project with more the 1000 watches and more that 10 developers according to the specification of the GHTorrent dump. We exclude the repository bytedeco/javacpp-presets as an outlier causing LISA to run into page faults. We excluded one repository requiring credentials. DJ-Rex hits the size limit for serialized results on repository SonarSource/sonarqube and wildfly/wildfly (29.000 and 28.000 commits), both repositories remaining in our sample as we are interested in the performance of the other approaches.

E. Correctness

The correctness of a Topleet variant can be checked during any processing step by comparing the index assignments with corresponding assignments produced by running the same task on a very basic reference implementation, i.e., a local and non-incremental realization of the core operations. We check such correspondence for different tasks on the repository sample.

F. Time

Since LISA is not distributed, we compare all approaches when running on a single machine. The averaged time needed for all 98 repositories by a solution is shown in Fig. 4. The averaged time needed for repositories grouped into exponentially growing commit count buckets is depicted in Fig. 3a. DJ-Rex misses time measurements for two repositories.

For repositories with low commit counts, all solution depicted in Fig. 3a need a comparable amount of time. The time increases differently with increasing commit count. The approaches are ranked as follows: DJ-Rex takes the most time (on average 67.12 minutes) followed by Topleet Index Based without memoization (10.61), LISA (6.44), DJ-Rex Optimized (5.4) and Topleet (1.33). DJ-Rex, DJ-Rex Optimized and Topleet Index Based do not employ memoization. This appears to be beneficial for low commit count but hampers the performance on high commit counts when computations tend to reoccurred – reflected by a bend in the average time at commit count 2990, faced for non-memoizing solutions.

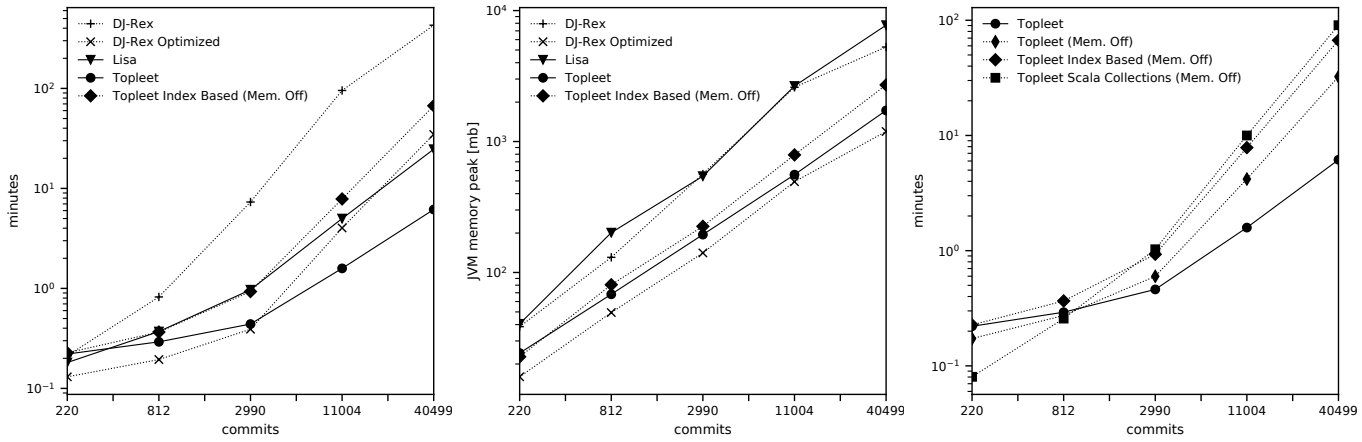
G. Memory

The comparison of the maximum amount of memory used in the JVM is done analogue to the comparison of time and depicted in Fig. 3b and Fig. 4. The memory peak is averaged over several repositories minimizing the influence of garbage collection.

Memory usage increases with rising commit count. For the average memory peak over all repositories, DJ-Rex Optimized (0.32 GB) is sightly better than Topleet (0.44), both running on Apache Spark in local mode. Topleet Index Based Memoization

[6]Commits included in branches can be flattened into one linear sequence.

[7]http://ghtorrent.org/downloads.html



(a) Time in minutes (**log-log scale**) (b) JVM Memory peak in mb (**log-log scale**) (c) Infrastructure in minutes (**log-log scale**)

Fig. 3. Evaluation applied to the 98 repositories averaged over five exponentially growing commit count buckets. Solutions not using memoization are depicted on a dotted line as such technique changes the performance characteristics.

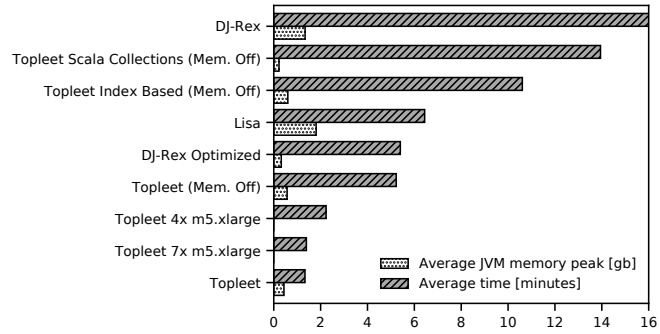


Fig. 4. The average characteristics of all approaches running on the 98 repositories, sorted by time and cut at 16 minutes (DJ-Rex needs 67 minutes). The memory profile for distributed solutions is excluded.

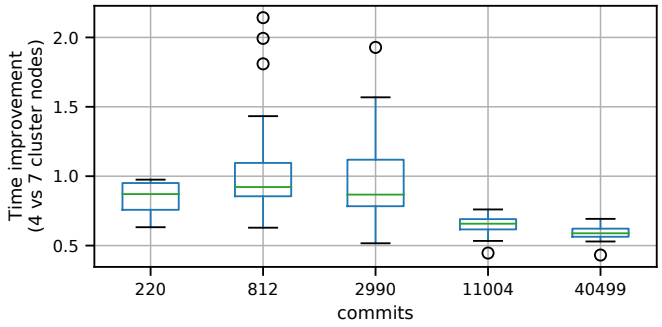


Fig. 5. Distribution: The time improvement running a solution on 7 cluster nodes (7x m5.xlarge) instead of running it on 4 cluster nodes (4x m5.xlarge). Both solutions include one master node.

Off (0.6) does not produce high overhead. DJ-Rex (1.34) and Lisa (1.81) both use the most memory.

H. Computation Infrastructure

We compare the application time using different Topleet infrastructure features in Fig. 3c. The Scala Collections API solution is best on low commit counts because it does not boot Apache Spark. For high commit counts, Topleet with memoization is the best (1.33 minutes on average), no memoization increases the time (5.23), followed by the index based variant that requires the index traversal (10.61) and the Scala collections realization (13.94).

I. Distribution

Increasing the number of core nodes in a distributed setup from 3 to 6 (i.e., Topleet 4x m5.xlarge and 7x m5.xlarge) changes the average time needed for repositories from 2.23 to 1.39 minutes. Fig. 5 shows the improvement ratio for the different commit buckets. For repositories with low commit counts, the improvement is low, including a high variation; adding hardware might even slow down the processing. For repositories with high commit count, the improvement is definite, ranging in an area around 60% of the original costs on a cluster with 3 core nodes.

VIII. Related Work

The following section discusses the related work from a MSR and non-MSR specific viewpoint; we include an overview on selected MSR approaches in TABLE I.

A. Analysis Languages and Interfaces

Non-MSR: Languages and interfaces used for writing data analysis are diverse. Map-reduce is recently one of the most popular functional interfaces [46]; iterative large scale graph computations can be expressed in Pregel [47] or Signal/Collect [39]; other applications involve writing graph queries [48] and rule based production systems [49]; data analysis on streams often formulates processing in terms of dataflow abstractions, such as sources, sinks, and stores [50]; applications working with relations, e.g., [18], [51], use corresponding query languages and interfaces, like SQL or Pandas.

MSR: The previous languages and interfaces can potentially be adapted to MSR. The DJ-Rex solution [30], [31] motivates the migration of existing technology to map-reduce; Boa [32], [33], [34] uses a DSL focused on visiting the Java ASTs in distributed manner; LISA [38], [37], [7] enables the usage of Signal/Collect [39] to analyses the history of an AST created by a plugged parsing technology. Previous work of us uses production rule systems to recover technology usage [52]. However, repositories are also processed using command line tools [53]. Topleet provides map-reduce syntax.

B. Parallel and Distributed Analysis

Non-MSR: Scalability is a general problem faced in data analysis. Map-reduce frameworks provide distributed and parallel computations [46]. Pregel [47] is optimized with respect to iterative graph computations, but comparable to map-reduce in its distribution. Signal/Collect [39] shares a stronger relation to actor frameworks facilitating asynchronous messaging. Actors systems can be distributed [54]. Stream processing can be distributed [55]. The processing of relations can be distributed [56].

MSR: DJ-Rex and Boa are both executed on a distributed map-reduce framework; LISA applies local parallelization, not employing that the distribution of Signal/Collect [39] is possible. Topleet delegates its core operations to a distributed map-reduce framework and thereby inherits distribution.

C. Incrementalization and Reduction of Redundancies

Non-MSR: Most of the general approaches provide extensions for incrementalization and redundancy reduction. Both strategies are employed in the batch processing platform DryadInc [57], reusing identical computations and processing changes. Closely related works are [58], [59], [60]. Systems sharing a corresponding understanding of abstract algebra, such as monoid structures and monoid homomorphisms, are [61], [50], [62]. The systems [50], [62] concern incremental stream processing. In the data base literature, incrementalization occurs as view maintenance [63], [64], [65]; production systems are incrementalized in [49]. Non MSR work describing the usage of abelian groups for incrementalization can be found in [40] for view maintenance and [41] for incrementalization by program transformation. In [66], aggregations related to our index processing are discusses. The authors of [67] introduce a formalization based on Adjunctions.

MSR: Topleet does not use monoids but Abelian groups, extending monoids by the inverse operator to capture deletion in the repository history, inspired by [40], [41]. Currently there is no approach that enables incremental processing of repository history. LISA covers the reduction of redundancies without employing insights on abstract algebra. We also reduce redundancies by enabling optional memoization mechanisms.

D. History Model

Non-MSR: Approaches to incrementalization share a corresponding understanding of how data evolves. Online approaches, like those concerned with stream processing [62], focus on low-latency by processing arriving changes. Offline processing, such as [57], [58], [59], [60], employ the evolution of fully available data to decrease computation costs. Some hybrid approaches switch between both [50].

TABLE I
MSR APPROACHES WITH GENERIC (●) AND RESTRICTED (○) PROCESSING MODELS. THE ABBREVIATIONS REFER TO: HISTORY MODEL, DISTRIBUTION, PARALLELIZATION, REDUNDANCY REDUCTION AND INCREMENTALIZATION.

MSR Approach	His.	Dis.	Par.	Red.	Inc.	Language & Interface
DJ-Rex	○	○	○	○	○	Map-reduce
Boa	○	●	●		○	Boa DSL
LISA	●		○	●	○	Signal/Collect
Topleet	●	●	●	●	●	Map-reduce

MSR: Online processing of streams in MSR has been discussed in [68] with respect to real time capabilities, adequate query models and data summarization techniques. A relevant stream-based method is GHTorent [51], gathering meta-data from GitHub's push API. Formalizing patches, merges and conflicts has been done in [69], [70]. Such works provide formalization rather than processing mechanisms. MSR approaches like DJ-Rex, LISA and Boa can be considered as offline. DJ-Rex and Boa do not make any strong assumptions on the underlying history model; LISA is limited to a linear history. Our work is the first that provides optimized offline processing of data evolving over the acyclic repository history.

IX. CONCLUSION

In this paper, we present an incremental map-reduce approach for mining repository history that is based on Abelian groups, group homomorphisms and indexing. We discuss the interchangeable processing of abstractions of resources and abstractions of resource changes in terms of mapping, filtering, group-wise aggregation and joining. We compare our approach to LISA and DJ-Rex. Topleet outperforms both in term of time needed to process a repository, uses less memory than LISA and does not involve manual optimizations concerned with change, as required for DJ-Rex. We used Apache Spark to distribute the processing causing a modest overhead.

Ongoing work includes the migration of earlier MSR studies into the Topleet infrastructure. We will try to integrate other processing mechanisms, focusing on the history as a whole, such as authorship attribution, refactoring detection and graph queries to the history.

REFERENCES

[1] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess," in *MSR*. ACM, 2011, pp. 153–162.

[2] M. Capraro, M. Dorner, and D. Riehle, "The patch-flow method for measuring inner source collaboration," in *MSR*. ACM, 2018, pp. 515–525.

[3] T. Fritz, J. Ou, G. C. Murphy, and E. R. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *ICSE (1)*. ACM, 2010, pp. 385–394.

[4] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse, "How Developers Drive Software Evolution," in *IWPSE*. IEEE Computer Society, 2005, pp. 113–122.

[5] G. Canfora, L. Cerulo, and M. D. Penta, "Identifying Changed Source Code Lines from Version Repositories," in *MSR*. IEEE Computer Society, 2007, p. 14.

[6] D. S. Kolovos, N. D. Matragkas, I. Korkontzelos, S. Ananiadou, and R. F. Paige, "Assessing the Use of Eclipse MDE Technologies in Open-Software Projects," in *OSS4MDE@MoDELS*, ser. CEUR Workshop Proceedings, vol. 1541. CEUR-WS.org, 2015, pp. 20–29.

[7] C. V. Alexandru, S. Panichella, S. Proksch, and H. C. Gall, "Redundancy-free analysis of multi-revision software artifacts," *Empirical Software Engineering*, vol. 24, no. 1, pp. 332–380, 2019.

[8] K. Aggarwal, A. Hindle, and E. Stroulia, "Co-evolution of project documentation and within github," in *MSR*. ACM, 2014, pp. 360–363.

[9] M. Lungu, M. Lanza, and O. Nierstrasz, "Evolutionary and collaborative software architecture recovery with Softwarenaut," *Sci. Comput. Program.*, vol. 79, pp. 204–223, 2014.

[10] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou, "Inferring Repository File Structure Modifications Using Nearest-Neighbor Clone Detection," in *WCRE*. IEEE Computer Society, 2012, pp. 325–334.

[11] G. Antoniol, M. D. Penta, and E. Merlo, "An Automatic Approach to identify Class Evolution Discontinuities," in *IWPSE*. IEEE Computer Society, 2004, pp. 31–40.

[12] X. Meng, B. P. Miller, W. R. Williams, and A. R. Bernat, "Mining Software Repositories for Accurate Authorship," in *ICSM*. IEEE Computer Society, 2013, pp. 250–259.

[13] T. Schmorleiz and R. Lämmel, "Similarity management of 'cloned and owned' variants," in *SAC*. ACM, 2016, pp. 1466–1471.

[14] J. Härtel, H. Aksu, and R. Lämmel, "Classification of APIs by hierarchical clustering," in *ICPC*. ACM, 2018, pp. 233–243.

[15] P. Salza, F. Palomba, D. D. Nucci, C. D'Uva, A. D. Lucia, and F. Ferrucci, "Do developers update third-party libraries in mobile apps?" in *ICPC*. ACM, 2018, pp. 255–265.

[16] A. C. Hora and M. T. Valente, "apiwave: Keeping track of API popularity and migration," in *ICSME*. IEEE Computer Society, 2015, pp. 321–323.

[17] J. Ossher, S. K. Bajracharya, and C. V. Lopes, "Automated dependency resolution for open source software," in *MSR*. IEEE Computer Society, 2010, pp. 130–140.

[18] S. Raemaekers, A. van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *MSR*. IEEE Computer Society, 2013, pp. 221–224.

[19] A. T. T. Ying, G. C. Murphy, R. T. Ng, and M. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Trans. Software Eng.*, vol. 30, no. 9, pp. 574–586, 2004.

[20] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *ICSE*. IEEE Computer Society, 2004, pp. 563–572.

[21] H. H. Kagdi, S. Yusuf, and J. I. Maletic, "Mining sequences of changed-files from version histories," in *MSR*. ACM, 2006, pp. 47–53.

[22] T. Molderez, R. Stevens, and C. D. Roover, "Mining change histories for unknown systematic edits," in *MSR*. IEEE Computer Society, 2017, pp. 248–256.

[23] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE*. IEEE, 2009, pp. 78–88.

[24] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller, "Predicting Faults from Cached History," in *ICSE*. IEEE Computer Society, 2007, pp. 489–498.

[25] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *MSR*. IEEE Computer Society, 2010, pp. 31–41.

[26] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ICSE*. ACM, 2008, pp. 181–190.

[27] G. Gousios, M. Pinzger, and A. van Deursen, "An exploratory study of the pull-based software development model," in *ICSE*. ACM, 2014, pp. 345–355.

[28] M. M. Rahman, C. K. Roy, and J. A. Collins, "CoRReCT: code reviewer recommendation in GitHub based on cross-project and technology experience," in *ICSE (Companion Volume)*. ACM, 2016, pp. 222–231.

[29] F. Rahman and P. T. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *ICSE*. ACM, 2011, pp. 491–500.

[30] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan, "MapReduce as a general framework to support research in Mining Software Repositories (MSR)," in *MSR*. IEEE Computer Society, 2009, pp. 21–30.

[31] W. Shang, B. Adams, and A. E. Hassan, "An experience report on scaling tools for mining software repositories using MapReduce," in *ASE*. ACM, 2010, pp. 275–284.

[32] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: Ultra-Large-Scale Software Repository and Source-Code Mining," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 7:1–7:34, 2015.

[33] ——, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories," in *ICSE*. IEEE Computer Society, 2013, pp. 422–431.

[34] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of APIs in large-scale code corpus," in *SIGSOFT FSE*. ACM, 2014, pp. 166–177.

[35] H. Cai and J. Jenkins, "Leveraging historical versions of Android apps for efficient and precise taint analysis," in *MSR*. ACM, 2018, pp. 265–269.

[36] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *ICSE*. ACM, 2018, pp. 483–494.

[37] C. V. Alexandru and H. C. Gall, "Rapid Multi-Purpose, Multi-Commit Code Analysis," in *ICSE (2)*. IEEE Computer Society, 2015, pp. 635–638.

[38] C. V. Alexandru, S. Panichella, and H. C. Gall, "Reducing redundancies in multi-revision code analysis," in *SANER*. IEEE Computer Society, 2017, pp. 148–159.

[39] P. Stutz, A. Bernstein, and W. W. Cohen, "Signal/Collect: Graph Algorithms for the (Semantic) Web," in *International Semantic Web Conference (1)*, ser. Lecture Notes in Computer Science, vol. 6496. Springer, 2010, pp. 764–780.

[40] D. Gluche, T. Grust, C. Mainberger, and M. H. Scholl, "Incremental Updates for Materialized OQL Views," in *DOOD*, ser. Lecture Notes in Computer Science, vol. 1341. Springer, 1997, pp. 52–66.

[41] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann, "A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation," in *PLDI*. ACM, 2014, pp. 145–155.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *NSDI*. USENIX Association, 2012, pp. 15–28.

[43] R. Bose and J. Frew, "Lineage retrieval for scientific data processing: a survey," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 1–28, 2005.

[44] J. Cheney, L. Chiticariu, and W. C. Tan, "Provenance in Databases: Why, How, and Where," *Foundations and Trends in Databases*, vol. 1, no. 4, pp. 379–474, 2009.

[45] G. Gousios and D. Spinellis, "GHTorrent: Github's data from a firehose," in *MSR*. IEEE Computer Society, 2012, pp. 12–21.

[46] K. Lee, Y. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with MapReduce: a survey," *SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2011.

[47] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD Conference*. ACM, 2010, pp. 135–146.

[48] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An Evolving Query Language for Property Graphs," in *SIGMOD Conference*. ACM, 2018, pp. 1433–1445.

[49] C. Forgy, "Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem," *Artif. Intell.*, vol. 19, no. 1, pp. 17–37, 1982.

[50] P. O. Boykin, S. Ritchie, I. O'Connell, and J. J. Lin, "Summingbird: A Framework for Integrating Batch and Online MapReduce Computations," *PVLDB*, vol. 7, no. 13, pp. 1441–1451, 2014.

[51] G. Gousios, "The GHTorent dataset and tool suite," in *MSR*. IEEE Computer Society, 2013, pp. 233–236.

[52] J. Härtel, M. Heinz, and R. Lämmel, "EMF Patterns of Usage on GitHub," in *ECMFA*, ser. Lecture Notes in Computer Science, vol. 10890. Springer, 2018, pp. 216–234.

[53] D. Spinellis and G. Gousios, "How to analyze git repositories with command line tools: we're not in kansas anymore," in *ICSE (Companion Volume)*. ACM, 2018, pp. 540–541.

[54] G. A. Agha, *ACTORS - a model of concurrent computation in distributed systems*, ser. MIT Press series in artificial intelligence. MIT Press, 1990.

[55] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik, "Scalable Distributed Stream Processing," in *CIDR*. www.cidrdb.org, 2003.

[56] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL:

Relational Data Processing in Spark," in *SIGMOD Conference*. ACM, 2015, pp. 1383–1394.

[57] L. Popa, M. Budiu, Y. Yu, and M. Isard, "DryadInc: Reusing Work in Large-scale Computations," in *HotCloud*. USENIX Association, 2009.

[58] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," in *SoCC*. ACM, 2010, pp. 51–62.

[59] C. Yan, X. Yang, Z. Yu, M. Li, and X. Li, "IncMR: Incremental Data Processing Based on MapReduce," in *IEEE CLOUD*. IEEE Computer Society, 2012, pp. 534–541.

[60] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini, "Incoop: MapReduce for incremental computations," in *SoCC*. ACM, 2011, p. 7.

[61] M. Hayes and S. Shah, "Hourglass: A library for incremental processing on Hadoop," in *BigData*. IEEE Computer Society, 2013, pp. 742–752.

[62] L. Fegaras, "Incremental Query Processing on Big Data Streams," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 11, pp. 2998–3012, 2016.

[63] R. Chirkova and J. Yang, "Materialized Views," *Foundations and Trends in Databases*, vol. 4, no. 4, pp. 295–405, 2012.

[64] D. Quass, A. Gupta, I. S. Mumick, and J. Widom, "Making Views Self-Maintainable for Data Warehousing," in *PDIS*. IEEE Computer Society, 1996, pp. 158–169.

[65] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining Views Incrementally," in *SIGMOD Conference*. ACM Press, 1993, pp. 157–166.

[66] L. Libkin and L. Wong, "Query Languages for Bags and Aggregate Functions," *J. Comput. Syst. Sci.*, vol. 55, no. 2, pp. 241–272, 1997.

[67] J. Gibbons, F. Henglein, R. Hinze, and N. Wu, "Relational algebra by way of adjunctions," *PACMPL*, vol. 2, no. ICFP, pp. 86:1–86:28, 2018.

[68] G. Gousios, D. Safaric, and J. Visser, "Streaming software analytics," in *BIGDSE@ICSE*. ACM, 2016, pp. 8–11.

[69] C. Angiuli, E. Morehouse, D. R. Licata, and R. Harper, "Homotopical patch theory," in *ICFP*. ACM, 2014, pp. 243–256.

[70] S. Mimram and C. D. Giusto, "A Categorical Theory of Patches," *Electr. Notes Theor. Comput. Sci.*, vol. 298, pp. 283–307, 2013.