Provenance Annotation and Analysis to Support Process Re-Computation

Jacek Cała[0000−0002−8322−4370] and Paolo Missier[0000−0002−0978−2446]

School of Computing, Newcastle University, Newcastle upon Tyne, UK,
{Jacek.Cala, Paolo.Missier}@ncl.ac.uk

Abstract. Many resource-intensive analytics processes evolve over time following new versions of the reference datasets and software dependencies they use. We focus on scenarios in which any version change has the potential to affect many outcomes, as is the case for instance in high throughput genomics where the same process is used to analyse large cohorts of patient genomes, or *cases*. As any version change is unlikely to affect the entire population, an efficient strategy for restoring the currency of the outcomes requires first to identify the *scope of a change*, i.e., the subset of affected data products. In this paper we describe a generic and reusable provenance-based approach to address this scope discovery problem. It applies to a scenario where the process consists of complex hierarchical components, where different input cases are processed using different version configurations of each component, and where separate provenance traces are collected for the executions of each of the components. We show how a new data structure, called a *restart tree*, is computed and exploited to manage the change scope discovery problem.

Keywords: provenance annotations, process re-computation

1 Introduction

Consider data analytics processes that exhibit the following characteristics. C1: are resource-intensive and thus expensive when repeatedly executed over time, i.e., on a cloud or HPC cluster; C2: require sophisticated implementations to run efficiently, such as workflows with a nested structure; C3: depend on multiple reference datasets and software libraries and tools, some of which are versioned and evolve over time; C4: apply to a possibly large population of input instances, and C5: deliver valuable knowledge.

This is not an uncommon set of characteristics. A prime example is data processing for high throughput genomics, where the genomes (or exomes) of a cohort of patient cases are processed, individually or in batches, to produce lists of variants (genetic mutations) that form the basis for a number of diagnostic purposes (C5). These *variant calling and interpretation* pipelines take batches of 20–40 patient exomes and require hundreds of CPU-hours to complete (C1). Initiatives like the 100K Genome project in the UK (www.genomicsengland.co.uk) provide a perspective on the scale of the problem (C4).

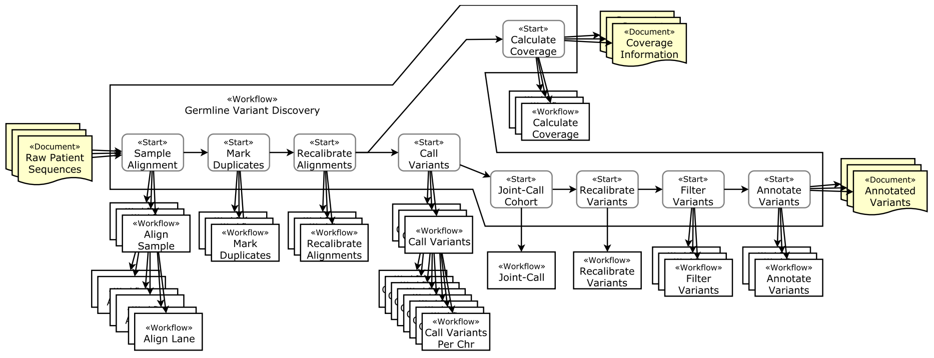


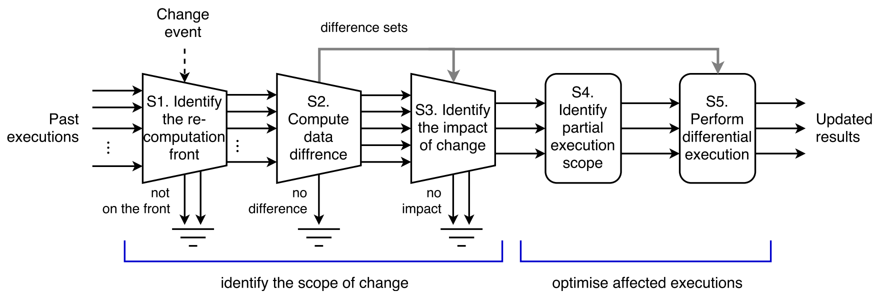
Fig. 1: A typical variant discovery pipeline processing a pool of input samples. Each step is usually implemented as a workflow or script that combines a number of tools run in parallel.

Figure 1, taken from our prior work [4], shows the nested workflow structure (C2) of a typical variant calling pipeline based on the GATK (Genomics Analysis Toolkit) best practices from the Broad Institute.[1] Each task in the pipeline relies on some GATK (or other open source) tool, which in turn requires lookups in public reference datasets. For most of these processes and reference datasets new versions are issued periodically or on an as-needed basis (C3). The entire pipeline may be variously implemented as a HPC cluster script or workflow. Each single run of the pipeline creates a hierarchy of executions which are distributed across worker nodes and coordinated by the orchestrating top-level workflow or script (cf. the "Germline Variant Discovery" workflow depicted in the figure).

Upgrading one or more of the versioned elements risks invalidating previously computed knowledge outcomes, e.g. the sets of variants associated with patient cases. Thus, a natural reaction to a version change in a dependency is to upgrade the pipeline and then re-process all the cases. However, as we show in the example at the end of this section, not all version changes affect each case equally, or in a way that completely invalidates prior outcomes. Also, within each pipeline execution only some of the steps may be affected. We therefore need a system that can perform more selective re-processing in reaction to a change. In [5] we have described our initial results in developing such a system for selective re-computation over a population of cases in reaction to changes, called `ReComp`. `ReComp` is a *meta-process* designed to detect the scope of a single change or of a combination of changes, estimate the impact of those changes on the population in scope, prioritise the cases for re-processing, and determine the minimal amount of re-processing required for each of those cases. Note that, while ideally the process of upgrading P is controlled by `ReComp`, in reality we must also account for upgrades of P that are performed "out-of-band" by developers, as we have assumed in our problem formulation.

[1] https://software.broadinstitute.org/gatk/best-practices

Briefly, ReComp consists of the macro-steps indicated in Fig. 2. The work presented in this paper is instrumental to the ReComp design, as it addresses the very first step (S1) indicated in the figure, in a way that is generic and agnostic to the type of process and data.



Change event
difference sets
Past executions
S1. Identify the re-computation front
S2. Compute data diffrence
S3. Identify the impact of change
S4. Identify partial execution scope
S5. Perform differential execution
Updated results
not on the front
no difference
no impact
identify the scope of change
optimise affected executions

Fig. 2: Schematic of the ReComp meta-process.

1.1 Version changes and their scope

To frame the problem addressed in the rest of the paper, we introduce a simple model for version changes as triggers for re-computation. Consider an abstract process P and a population $X = \{x_1 \dots x_N\}$ of inputs to P, referred to as *cases*. Let $\mathcal{D} = [D_1 \dots D_m]$ be an ordered list of *versioned* dependencies. These are components, typically software libraries or reference data sets, which are used by P to process a case. Each D has a version, denoted $D.v$, with a total order on the sequence of versions $D.v < D.v + 1 < \dots$ for each D.

An *execution configuration* for P is the vector $V = [v_1 \dots v_m]$ of version numbers for $[D_1, D_2, D_m]$. Typically, these are the latest versions for each D, but configurations where some D is "rolled back" to an older version are possible. The set of total orders on the versions of each $D \in \mathcal{D}$ induce a partial order on the set of configurations:

$$[v_1 \dots v_m] \prec [v'_1 \dots v'_m] \text{ iff } \{v_i \leq v'_i\}_{i:1 \dots m} \text{ and } v_i < v'_i \text{ for at least one } v_i.$$

We denote an *execution* of P on input $x_i \in X$ using configuration V by

$$E = P(x, V). \tag{1}$$

P may consist of multiple components $\{P_1 \dots P_k\}$, such as those in our example pipeline. When this is the case, we assume for generality that one execution $P(x, V)$ given x and V is realised as a collection $\{E_i = P_i(x, V)\}_{i:1 \dots k}$ of separate executions, one for each P_i. We use the W3C PROV [12] and ProvONE [6] abstract vocabularies to capture this model: $P, P_1 \dots P_k$ are all instances of `provone:Program`, their relationships is expressed as

$$\{\texttt{provone:hasSubProgram}(P, P_i)\}_{i:1 \dots k}$$

and each execution E_i is associated with its program P_i:

$$\{\texttt{wasAssociatedWith}(E_i, \_, P_i)\}_{i:1\dots k}$$

Version change events. We use PROV derivation statements `prov:wasDerived-From` to denote a *version change event* C for some D_i, from v_i to v'_i : $C = \{D.v'_i \xrightarrow{\text{wDF}} D.v_i\}$. Given $V = [v_1 \dots v_i \dots v_m]$, C *enables* the new configuration $V' = [v_1 \dots v'_i \dots v_m]$, meaning that V' can be *applied* to P, so that its future executions are of form $E = P(x, V')$.

We model sequences of changes by assuming that an unbound stream of change events $C_1, C_2, \dots$ can be observed over time, either for different or the same D_i. A re-processing system may react to each change individually. However, we assume the more general model where a set of changes accumulate into a window (according to some criteria, for instance fixed-time) and is processed as a batch. Thus, by extension, we define a composite change to be a set of elementary changes that are part of the same window. Given $V = [v_1 \dots v_i \dots v_j \dots v_m]$, we say that $C = \{D.v'_i \xrightarrow{\text{wDF}} D.v_i, D.v'_j \xrightarrow{\text{wDF}} D.v_j, \dots\}$ enables configuration $V' = [v_1 \dots v'_i \dots v'_j \dots v_m]$. Importantly, all change events, whether individual or accumulated into windows, are merged together into the single *change front* CF which is the configuration of the latest versions of all changed artefacts.

Applying CF to $E = P(x, V)$ involves re-processing x using P to bring the outcomes up-to-date with respect to all versions in the change front. For instance, given $V = [v_1, v_2, v_3]$ and the change front $CF = \{v'_1, v'_2\}$, the new execution of $E = P(x, [v_1, v_2, v_3])$ is $E' = P(x, [v'_1, v'_2, v_3])$. Note that it is important to keep track of how elements of the change front are updated as it may be possible to avoid re-executing some of P's components for which the configuration has not changed. Without this fine-grained derivation information, each new execution for each x simply uses the latest versions but cannot be optimised using partial re-processing.

Clearly, processing change events as a batch is more efficient than processing each change separately, e.g. $E' = P(x, [v'_1, v_2])$ followed by $E'' = P(x, [v'_1, v'_2])$. But a model that manages change events as a batch is also general in that it accommodates a variety of refresh strategies. For example, applying changes that are known to have limited impact on the outcomes can be delayed until a sufficient number of other changes have accumulated into CF, or until a specific high-impact change event has occurred. A discussion of specific strategies that are enabled by our scope discovery algorithm is out of the scope of this paper.

1.2 Problem formulation and contributions

Suppose P has been executed h times for some $x \in X$, each time with a different configuration $V_1 \dots V_h$. The collection of past executions, for each $x \in X$, is:

$$\{E(P_i, x, V_j)_{i:1\dots k, j:1\dots h, x \in X}\} \tag{2}$$

The problem we address in this paper is to identify, for each change front CF, the smallest set of those executions that are affected by C. We call this the *re-computation front* C relative to X. We address this problem in a complex general setting where multiple types of time-interleaved changes are allowed, where multiple configurations are enabled by any of these changes, and where executions may reflect any of these configurations, and in particular individual cases x may be processed using any such different configurations. The example from the next section illustrates how this setting can manifest itself in practice.

Our main contribution is an algorithm for discovering re-computation fronts that applies generically to a range of processes, from simple black-box, single component workflows where P is indivisible, to complex hierarchical workflows where P consists of components P_i, and each of these may itself be defined in terms of sub-components.

Following a tradition from the literature to use provenance as a means to address re-computation [2,10,5], our approach also involves collecting and exploiting both execution provenance for each E, as well as elements of process–subprocess dependencies as mentioned above. To the best of our knowledge this particular use of provenance and the algorithm have not been proposed before.

1.3 Example: versioning in Genomics

The problem of version change emerges concretely in Genomics pipelines in which changes have different scope, both within each process instance and across the population of cases. For example, an upgrade to the `bwa` aligner tool directly affects merely the alignment task but its impact may propagate to most of the tasks downstream. Conversely, an upgrade in the human reference genome directly affects the majority of the tasks. In both cases, however, the entire population of executions is affected because current alignment algorithms are viewed as "black boxes" that use the entire reference genome.

On the other hand, a change in one of the other reference databases that are queried for specific information, only affects those cases where some of the changed records are part of a query result. One example is ClinVar, a popular variants database which is queried to retrieve variants information about specific diseases (phenotypes). In this case, changes that concern one phenotype will not affect cases that exhibit a completely different phenotype.

Additionally, note that version changes in this example occur with diverse frequency. For instance, the reference genome is updated every few years, alignment libraries every few months, and ClinVar every month.

2 Recomputation fronts and restart trees

2.1 Recomputation fronts

In Sec. 1.1 we have introduced a partial order $V \prec V'$ between process configurations. In particular, given V, if a change C enables V' then by definition $V \prec V'$. Note that this order induces a corresponding partial order between any two executions that operate on the same $x \in X$.

$$E = P(x, V) \ll E' = P(x, V') \text{ iff } V \prec V' \tag{3}$$

This order is important, because optimising re-execution, i.e. executing $P(x, V')$, may benefit from the provenance associated with the previous execution according to the sequence of version changes that is $E = P(x, V)$ (a discussion on the precise types of such optimisations can be found in [5]). For this reason in our implementation we keep track of the execution order explicitly using the `wasInformedBy` PROV relationship, i.e. we record PROV statement $E' \xrightarrow{\text{wIB}} E$ whenever $E \ll E'$.

To see how these chains of ordered executions may evolve consider, for instance, $E_0 = P(x_1, [a_1, b_1]), E_1 = P(x_2, [a_1, b_1])$ for inputs x_1, x_2 respectively, where the a and b are versions for two dependencies D_1, D_2. The situation is depicted in Fig.3(left). When change $C_1 = \{a_2 \xrightarrow{\text{wDF}} a_1\}$ occurs, it is possible that only x_1 is re-processed, but not x_2. This may happen, for example, when D_1 is a data dependency and the change affects parts of the data which were not used by E_1 in the processing of input x_2. In this case, C would trigger one single new execution: $E_2 = P(x_1, [a_2, b_1])$ where we record the ordering $E_0 \ll E_2$. The new state is depicted in Fig. 3(middle).

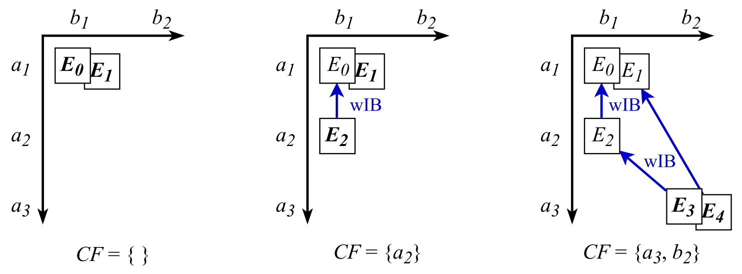


Fig. 3: The process of annotating re-execution following a sequence of events; in bold are executions on the re-computation front; a- and b-axis represent the artefact derivation; arrows in blue denote the `wasInformedBy` relation.

Now consider the new change $C_2 = \{a_3 \xrightarrow{\text{wDF}} a_2, b_2 \xrightarrow{\text{wDF}} b_1\}$, affecting both D_1 and D_2, and suppose both x_1 and x_2 are going to be re-processed. Then, for each x we retrieve the latest executions that are affected by the change, in this case E_2, E_1, as their provenance may help optimising the re-processing of x_1, x_2 using the new *change front* $\{a_3, b_2\}$. After re-processing we have two new executions: $E_3 = P(x_1, [a_3, b_2]), E_4 = P(x_2, [a_3, b_2])$ which may have been optimised using E_2, E_1, respectively, as indicated by their ordering: $E_3 \ll E_2$, $E_4 \ll E_1$.

To continue with the example, let us now assume that the provenance for a new execution: $E_5 = P(x_1, [a_1, b_2])$ appears in the system. This may have been triggered by an explicit user action independently from our re-processing system. Note that the user has disregarded the fact that the latest version is a_3. The corresponding scenario is depicted in Fig. 4(left). We now have two

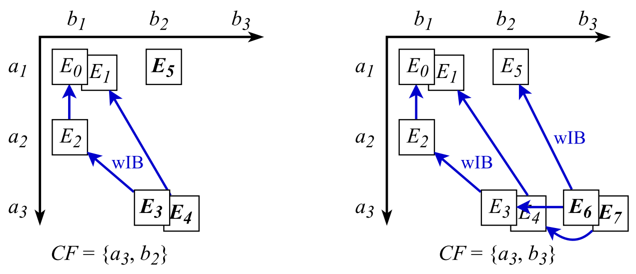


Fig. 4: Continuation of Fig. 3; in bold are executions on the re-computation front; a- and b-axis represent the artefact derivation; arrows in blue denote the `wasInformedBy` relation.

executions for x_1 with two configurations. Note that despite $E_0 \ll E_5$ holds it is not reflected by a corresponding $E_5 \xrightarrow{\text{wIB}} E_0$ in our re-computation system because E_5 was an explicit user action. However, consider another change event: $\{b_3 \xrightarrow{\text{wDF}} b_2\}$. For x_2, the affected executions is E_4, as this is the single latest execution in the ordering recorded so far for x_2. But for x_1 there are now two executions that need to be brought up-to-date, E_3 and E_5, as these are the maximal elements in the set of executions for x_1 relative according to the order: $E_0 \ll E_2 \ll E_3, E_0 \ll E_5$. We call these executions the *recomputation front* for x_1 relative to change front $\{a_3, b_3\}$, in this case.

This situation, depicted in Fig. 4 (right), illustrates the most general case where an entire set of previous executions need to be considered when re-processing an input with a new configuration. Note that the two independent executions E_3 and E_5 have merged into the new E_6.

Formally, the *recomputation front* for $x \in X$ and for a change front $CF = \{w_1 \dots w_k\}, k \leq m$ is the set of maximal executions $E = P(x, [v_1 \dots v_m])$ where $v_i \leq w_i$ for $1 \leq i \leq m$.

2.2 Building a Restart Tree

Following our goal to develop a generic re-computation meta-process, the front finding algorithm needs to support processes of various complexity – from the simplest black-box processes to complex hierarchical workflows mentioned earlier. This requirement adds another dimension to the problem of the identification of the re-computation front.

If process P has a hierarchical structure, which in Sec. 1.1 we have expressed using the `provone:hasSubProgram` statement, one run of P will usually result in a collection of executions. These are logically organised into a hierarchy, where the top-level represents the execution of the program itself, and sub-executions (connected via `provone:wasPartOf`) represent the executions of the sub-programs. Following the principle of the separation of concerns, we assume the general case where the top-level program is not aware of the data and software dependencies of all its parts. Thus, discovering which parts of the program used a particular dependency requires traversing the entire hierarchy of executions.

To illustrate this problem let us focus on a small part of the Genomics pipeline – the alignment step (Align Sample and Align Lane). Fig. 5 shows this step modelled using ProvONE. P_0 denotes the top program – the Align Sample workflow, SP_0 is the Align Lane subprogram, SSP_0–SSP_3 represent the subsub-programs of bioinformatic tools like `bwa` and `samtools`, while SP_1–SP_3 are the invocations of the `samtools` program. Programs have input and output ports (the dotted grey arrows) and ports p_1–p_8 are related with default artefacts a_0, b_0, etc. specified using the `provone:hasDefaultParam` statement. The artefacts refer to the code of the executable file and data dependencies; e.g. e_0 represents the code of `samtools`. Programs are connected to each other via ports and channels, which in the figure are identified using reversed double arrows.

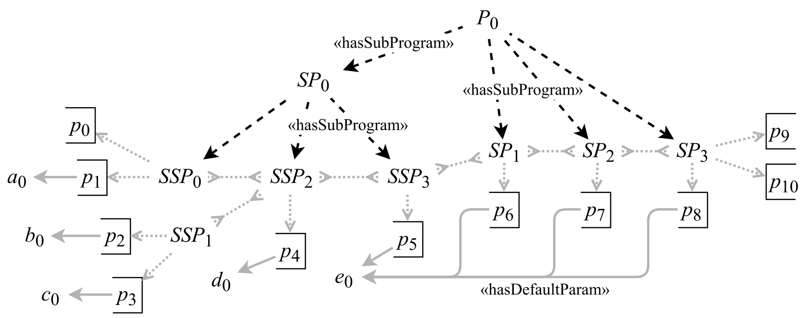


Fig. 5: The structure of a small part of the Genomics pipeline shown in Fig. 1 encoded in the ProvONE model. (- - ➤) – the `hasSubProgram` relation between programs; (⟶) – the `hasDefaultParam` statements; (⋯➤) – `hasInPort`/`hasOutPort`; (➤⋯<) – the sequence of the $\{P_i$ `hasOutPort` p_m `connectsTo` Ch_x, P_j `hasInPort` p_n `connectsTo` $Ch_x\}$ statements.

Running this part of the pipeline would generate the runtime provenance information with the structure resembling the program specification (cf. Fig. 6). The main difference between the static program model and runtime information is that during execution all ports transfer some data – either default artefacts indicated in the program specification, data provided by the user, e.g. input sample or the output data product. When introducing a change in this context, e.g. $\{b_1 \xrightarrow{\text{wDF}} b_0, e_1 \xrightarrow{\text{wDF}} e_0\}$, two things are important. Firstly, the usage of the artefacts is captured at the sub-execution level (SSE_1, SSE_3 and SE_1–SE_3) while E_0 uses these artefacts indirectly. Secondly, to rerun the alignment step it is useful to consider the sub-executions grouped together under E_0, which determines the end of processing and delivers data y_0 and z_0 meaningful for the user. We can capture both these elements using the tree structure that naturally fits the hierarchy of executions encoded with ProvONE. We call this tree the *restart tree* as it indicates the initial set of executions that need to be rerun. The tree also provides references to the changed artefacts, which is useful to perform further steps of the ReComp meta-process. Fig. 6 shows in blue the restart tree generated as a result of change in artefacts b and c.

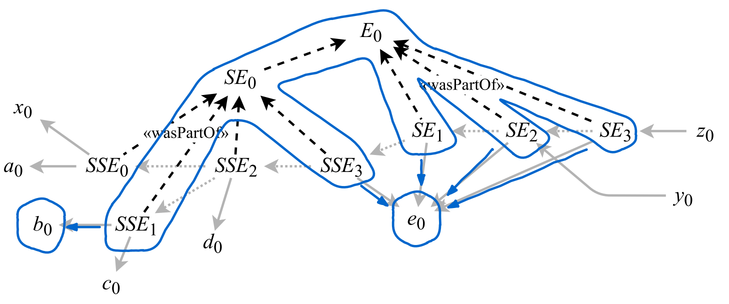


Fig. 6: An execution trace for the program shown in Fig. 5 with the restart tree and artefact references highlighted in blue. ($\dashrightarrow$) – the `wasPartOf` relation between executions; ($\longrightarrow$) – the `used` statements; ($\cdots\!\rightarrow$) – the sequence of the E_j `used` z `wasGeneratedBy` E_i statements.

Finding the restart tree is straightforward. It involves building paths from the executions that used artefacts that have changed, all the way up to the top-level execution following the `wasPartOf` relation. The tree is formed by merging all the paths that end with the same execution.

3 Computing The Re-computation Front

Combining together all three parts discussed above, we present in Listing 1.1 the pseudocode of our algorithm to identify the re-computation front. The input of the algorithm is the change front CF that the ReComp framework keeps updating with every change observed. The output is a list of restart trees, each rooted with the top-level execution. Every node of the tree is triple: (E, [*changedData*], [*children*]) that combines an execution with optional list of changed data artefacts it used and optional list of sub-executions it coordinated. For executions that represent a simple black-box process the output of the algorithm reduces to the list of triples like: $[(E_i, [a_k, a_l, \ldots], [\,]), (E_j, [a_m, a_n, \ldots], [\,]), \ldots]$ in which the third element of each node is always empty. For the example of a hierarchical process discussed in the previous section the output would be $[(E_0, [\,], [(SE_0, [\,], [(SSE_1, [b_0], [\,]), (SSE_3, [e_0], [\,])]), (SE_1, [e_0], [\,]), (SE_2, [e_0], [\,]), (SE_3, [e_0], [\,])])]$

The algorithm starts by creating the root node, OutTree, of an imaginary tree that will combine all independent executions affected by the change front.

Then, it iterates over all artefacts in the ChangeFront set. For each of them the algorithm traverses the chain of versions: Item $\xrightarrow{\text{wDF}}$ PredI $\xrightarrow{\text{wDF}}$ PPredI $\xrightarrow{\text{wDF}}$ ... (line 4), and then for each version it looks up all the executions that used particular version of the data (line 5). The core of the algorithm are lines 6–7 used to build the trees out of the affected executions. In line 6 a path from the affected execution to its top-level parent execution is built. Then, in line 7, the path is merged with the OutTree such that two paths with the same top-level execution are combined into the same subtree, whereas paths with different root execution will become two different subtrees on the OutTree.children list.

Listing 1.1: An algorithm to find the re-computation front.

```
1 function find_recomp_front(ChangeFront) : TreeList
2 OutTree := (root,data := [],children := [])
3 for Item in ChangeFront do
4 for PredI in traverse_derivations(Item) do
5 for Exec in iter_used(PredI) do
6 Path := path_to_root(PredI,Exec)
7 OutTree.merge_path(Path)
8
9 return OutTree.childern
```

Listing 1.2 shows the path_to_root function that creates the path from the given execution to its top-level parent execution. The function first checks whether or not the given execution Exec has already been re-executed (lines 4–6). It does so by iterating over all `wasInformedBy` statements in which Exec is the informant and checking if the statement is typed as recomp:re-execution. If there is such statement path_to_root returns the empty path to indicate that Exec is not on the re-execution front (line 6). Otherwise, if none of the communication statements indicates re-execution by ReComp, Exec is added to the path (line 9) and algorithm moves up to check the parent execution (line 8–10). This is repeated until Exec is the top-level parent in which case Y in line 8 becomes null and the repeat loop ends.

Listing 1.2: Function to generate the path from the given execution to its top-level parent.

```
1  function path_to_root(ChangedItem, Exec) : Path
2    OutPath := [ChangedItem]
3    repeat
4      for wIB in iter_was_informed_by(Exec)
5        if typeof(wIB) is "recomp:re-execution" then
6          return []
7      OutPath.append(Exec)
8      was_part_of(Exec, Y)
9      Exec := Y
10   until Exec = null
11   return OutPath
```

The discussion on other functions used in the proposed algorithm, such as traverse_derivations and iter_used, is omitted from the paper as they are simple technical operations. Interested readers can download the complete algorithm written in Prolog from our GitHub repository.[2]

4 Related Work

A recent survey by Herschel et al. [8] lists a number of applications of provenance such as improving collaboration, reproducibility and data quality. It does not highlight, however, the importance of process re-computation which we believe needs much more attention nowadays. Large, data-intensive and complex analytics requires effective means to refresh its outcomes while keeping the re-computation costs under control. This is the goal of the `ReComp` meta-process [5]. To the best of our knowledge no prior work addresses this or a similar problem.

Although some research on the use of provenance in re-computation was proposed earlier, it focused on the final steps of our meta-process: partial or differential re-execution. Altintas et al. discussed in [2] the "smart" rerun of workflows in Kepler, which can take into account data dependencies such that only the parts of a workflow affected by a change are re-executed. Starflow [3] allowed the workflow structure and subworkflow downstream the change to be automatically discovered using static, dynamic and user annotations. Ikeda et al. [9] proposed a solution to determine precisely the fragment of a data-intensive program that needed to be re-executed to refresh stale results. Also, Lakhani et al. [10] presented an approach for rollback and re-execution of a process.

The key difference between those efforts and our work is that we consider re-computation in the view of a whole population of past executions. Executions which may not even belong to the same data analysis. From such population, we select the executions which are affected by a change and for each of them we find the restart tree. Later, the tree may be used to initiate partial rerun.

Another use of provenance to track changes in scientific processes has been proposed by Freire et al. [7] and more recently by Pimentel et al. [13]. They address the problem of evolution of workflows/scripts, i.e. the changes in the process structure that affect the outcomes. The focus of their work is complementary to our view, however. They use provenance to understand what has changed and to help make a decision which execution provides the best results. We, instead, observe changes in the environment and only then react to them by finding the minimal set of executions that require refresh.

5 Discussion and Conclusions

In this paper we presented a generic approach to use provenance annotations to inform a re-computation framework about the selection of past execution that require refresh upon a change in their data and software dependencies. We call this selection the re-computation front. We have presented an algorithm for computing the front, which relies on annotations of re-executions to maintain the

[2] `https://www.github.com/???` – to be included in the final manuscript.

most up-to-date version of the dependencies, and to handle composite structure of processes.

Execution times for our implementation, written in Prolog, is of the order of milliseconds when run on a 250 MB database of provenance facts for about $\approx$ 56k composite executions and a set of artefact documents of which two had 15 and 19 version changes.

In line with [1], we note that a generic provenance capture facility which stores basic information about processes and data is often not enough to support the needs of applications. For our algorithm to work properly, we have to additionally annotate every re-execution with the `wasInformedBy` statement, so the past executions are not executed again multiple times. This indicates that the ProvONE model defines only a blueprint with minimal set of meta-information to be captured which needs to be extended within each application domain.

References

1. Alper, P., Belhajjame, K., Curcin, V., Goble, C.: LabelFlow Framework for Annotating Workflow Provenance. Informatics 5(1), 11 (2018)
2. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance collection support in the kepler scientific workflow system. In: Moreau, L., Foster, I. (eds.) Provenance and Annotation of Data. vol. 4145, 2006.
3. Angelino, E., Yamins, D., Seltzer, M.: Starflow: A script-centric data analysis environment. In: McGuinness, D.L., Michaelis, J.R., Moreau, L. (eds.) Provenance and Annotation of Data and Processes, 2010.
4. Cała, J., Marei, E., Xu, Y., Takeda, K., Missier, P.: Scalable and efficient whole-exome data processing using workflows on the cloud. Future Generation Computer Systems (jan 2016).
5. Cała, J., Missier, P.: Selective and recurring re-computation of Big Data analytics tasks: insights from a Genomics case study. Tech. Rep. School of Computing, Newcastle University, 2017.
6. Cuevas-Vicenttín, V., Ludäscher, B., Missier, P., Belhajjame, K. *et al.*: ProvONE: A PROV Extension Data Model for Scientific Workflow Provenance (2016).
7. Freire, J., Silva, C.T., Callahan, S.P., Santos, E., Scheidegger, C.E., Vo, H.T.: Managing Rapidly-evolving Scientific Workflows. Proceedings of the 2006 International Conference on Provenance and Annotation of Data, 2006
8. Herschel, M., Diestelkämper, R., Ben Lahmar, H.: A survey on provenance: What for? What form? What from? The VLDB Journal 26(6), 1–26 (2017)
9. Ikeda, R., Das Sarma, A., Widom, J.: Logical provenance in data-oriented workflows. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE).
10. Lakhani, H., Tahir, R., Aqil, A., Zaffar, F., Tariq, D., Gehani, A.: Optimized Rollback and Re-computation. In: 2013 46th Hawaii International Conference on System Sciences.
11. McSherry, F.D., Murray, D.G., Isaacs, R., Isard, M.: Differential Dataflow. In: 6th Biennial Conference on Innovative Data Systems Research (CIDR '13) (2013)
12. Moreau, L., Missier, P., Belhajjame, K., B'Far, R., Cheney, J. *et al.* : PROV-DM: The PROV Data Model. Tech. rep., World Wide Web Consortium (2013)
13. Pimentel, J.F., Murta, L., Braganholo, V., Freire, J.: noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts. Proceedings of the VLDB Endowment 10(12), 2017.