Technical report

Nr.: FIN-01-2012

A Hierarchical Framework for Provenance Based on Fragmentation and Uncertainty

Martin Schäler, Sandro Schulze, and Gunter Saake

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-01-2012

A Hierarchical Framework for Provenance Based on Fragmentation and Uncertainty

Martin Schäler, Sandro Schulze, and Gunter Saake

Arbeitsgruppe Datenbanken

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Martin Schäler
Postfach 4120
39016 Magdeburg
E-Mail: schaeler@iti.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 17.02.2012

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

A Hierarchical Framework for Provenance Based on Fragmentation and Uncertainty

Martin Schäler, Sandro Schulze, Gunter Saake
School of Computer Science
University of Magdeburg, Germany
{schaeler, sanschul, saake}@iti.cs.uni-magdeburg.de

Categories and Subject Descriptors
H.0 [**Information Systems**]: General

Keywords
Provenance Framework, Result Verification, Uncertainty

Abstract
In the recent past, provenance – a research field that can be used to determine the origin and derivation history of data – has gained much attention [12]. Additionally, provenance is an important field for validating computation results. It covers coarse grained data as well as very fine grained information showing details of the implementation. Moreover, it is highly related to different topics such as causality [25]. Unfortunately, there is currently no global framework covering existing approaches and addressing the versatile characteristics of provenance. In this paper, we suggest a hierarchical framework for provenance based on its most general and common characteristics w.r.t. to the current state of the art. In fact, the framework contains different layers of abstraction. We discuss the use and limitations of all layers by means of existing models and formalisms that lay the foundation for each layer. Additionally, we explain the relationship between these layers and analyze how existing approaches interact with our framework.

1. INTRODUCTION
What is provenance? As this term is used in many communities, such as relational databases [9, 14, 18], (scientific) workflows [27, 28, 30], and even to determine source code ownership [15], we cannot give an overall definition sufficient for all of these communities. Moreover, we even cannot give a clear definition to one of these communities as pointed out in [12], because of the diversity of provenance systems. But what we can say is what provenance is useful for: (1) Understanding (and possibly recomputation) of *foreign results* to validate or explain them and (2) Computation (and subsequent validation) of *own results* based on the provenance of previous results, which are used as input [12]. Moreover, several authors in the field of provenance have identified certain characteristics that seem to hold for provenance generally:

Unchangeability. Provenance describes what actually happened in the past and does not describe future alternatives. Therefore, it is unchangeable [12].

Fragmentation. Every provenance system has some starting point. As Braun argues in a provoking manner, to be complete, we have to be able to track back the history of any data item to the Big Bang [8]. Additionally, coarse grained provenance notions (e.g., for workflows) omit possibly important details of the derivation process [11]. In contrast, other forms of provenance contain a lot of very fine granular information, which might be hard to understand because the big picture is missing and there is simply to much information [1]. Thus, this is about the availability of (complete) provenance information at a certain level of granularity (fragmentation).

Uncertainty. Provenance always contains uncertainty to some extent. Thus, in most cases no provenance is better than wrong provenance [24]. This problem also becomes visible in several papers dealing with *secure provenance*, such as [22, 23, 24] showing the necessity for reliable provenance.

Recently, provenance gained much attention [9, 12, 18, 25]. Unfortunately, it is very difficult to assign single research results to challenges in provenance, relate them to different aspects of provenance, alternative solutions or evaluate advantages and (possibly not obvious) drawbacks of a certain approach, because there is no general provenance framework covering this topic. To address this problem, we suggest such a general framework for provenance. To ensure generality, our framework is based on the previously introduced general characteristics. In fact, we use differences regarding *fragmentation* and *uncertainty* to describe abstraction layers that can be used for different purposes and contain several open research challenges. Knowing that our framework might be incomplete and furthermore there might be use cases that are not covered within, we explicitly encourage the reader to question and extend the results of this paper. Particularly, we make the following contributions:

- We suggest a general hierarchical framework for provenance addressing fragmentation and uncertainty,
- show how existing formalisms and notations can be used for different layers of abstraction in the framework,

- discuss the applicability and limitations of equivalent formal approaches in different data models,
- present extensions of existing approaches to link the single layers of our framework,
- explain the use and limitations of granularity refinements within the layers.

2. BACKGROUND AND NOTATION

Subsequently, we introduce the notation of the *Open Provenance Model* (OPM) for coarse grained provenance and existing formalisms for fine grained provenance which are relevant for the remainder of this paper.

2.1 The Open Provenance Model

The OPM [27] is an example for coarse grained workflow provenance. According to the OPM, the causal dependencies of a *particular* data item within a provenance system are modeled as an acyclic graph (V, E). The vertices of this graph represent: (1) *Artifacts* naming physical objects or their digital representation, (2) Processes taking input artifacts to create new artifacts, or (3) Agents that have certain effects on processes. The edges of the provenance graph show the relationships between the vertices. For instance, Process P1 takes two input Artifacts of certain roles to create an *Artifact Result* and is controlled by *Agent A1* (Figure 1). As a result, Figure 1 shows the past derivation history of the *Result Artifact*, but not possible future usage of the artifacts or alternatives in the past in the OPM. Furthermore, we extended the OPM with additional semantics information. As depicted in Figure 1, we distinguish between three different artifact types: Initial data items, intermediate results, and final results[1]. This differentiation is important, because we do not produce initial data items in a provenance system and thus have to rely on (provenance) information tagged to this object (*fragmentation*). In contrast, we produce and consume intermediate results in our system. Moreover, final results may be initial data items for following (future) processes.

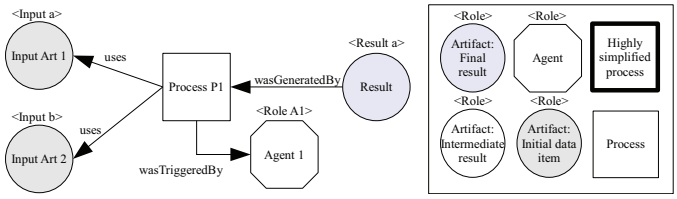


Figure 1: Provenance Graph

2.2 The Relationship between Lineage, Why, How, and Where Provenance

As already mentioned, we will build our provenance framework based on existing formalisms. Hence, we briefly introduce Lineage [14], Why [9], How [18] and Where [9] provenance formalisms and their relationships between each other according to the formalization of Cheney et al. for the relational data model [11]. To explain the relationships of these terms, we use the example in Figure 2. The example contains a database instance I with two binary relations $R(a, b)$ and $S(c, d)$. Furthermore, the SPJU[2] queries $q(x)$ and $r(x, y)$ create the views T and U respectively. Note that we annotated all tuples in I such as t_1 for the first tuple in R. In this example, we are interested in how t_8 was created. To do so, we track back the derivation history of t_8 to tuples in I. Particularly, we found two intentions of provenance:

(A) Existence. *Lineage, Why-*, and *How* provenance explain *why* a tuple is *part of the result* of a query (sequence). Therefore, these terms are important for recomputation and validation of query results.
(B) Value origin. However, the terms *Where* and (partially) *How* provenance show where *attribute values* stem from, respectively *how* are they computed if the tuple exists. Note that How provenance can explain why a tuple exists and in some cases this approach also denotes how values are computed as we will see shortly.
In the sequel, we will explain the basic principles and formalisms of Lineage, Why, How and Where provenance with the help of the example in Figure 2 and how the approaches relate to these two intentions.

Lineage. Formally, Lineage $Lin(Q, I, t_n)$ for a SPJU query Q, a DB I and a tuple t_n is a subset of I which is used to create t_n according to Q. For simplicity, we write $Lin(t_n)$ because we refer to the example DB instance and queries. Note that there might be several paths to compute t_n because of the set semantics of the relational model. Thus, a tuple t_k occurs only once in Lin although it might be used in several paths (e.g., as join partner). In the example, $Lin(t_8) = \{t_4, t_6\}$ expresses that these tuples are somehow used to create t_8 without showing any details such as join partners.

Why Provenance. In contrast to Lineage, Why provenance $Why(t_n)$ contains a *set of sets* denoting the paths (similar to the routes in [13]) that indicate *why* a tuple in the output was created, which is not shown in Lineage. For t_6 in the example, we see that are two possible paths to create t_6: Joining t_1 and t_4, or joining t_2 and t_5, because of the set semantics. With bag semantics, every $Why(t_{6_n})$ would contain exactly one path. Furthermore, Cheney et al. have shown that Lineage is an instance of Why provenance [11].

How Provenance. As in Why provenance, How provenance $How(t_n)$ denotes the paths that indicate *how* a tuple was created in the notion of polynomials, containing the operations + and ×. Theses operations are defined for a set K in a commutative semiring $(K, 0, 1, +, \times)$. Thus, this approach is also known as semiring model. Independent of the semantics of the operations, the polynomial shows the paths creating an output tuple such as $How(t_6) = (t_1 \times t_4) + (t_2 \times t_5)$. Dependent on the semantics of the semiring, it can be used to compute annotation values or even attribute values. For example, it is possible to compute the uncertainty value of tuples. Therefore, annotations such as t_8 contain the probability of this tuple in a probabilistic database and the polynomial is mapped to a probability semiring using a semiring homorphism [11, 18]. Recent work extended How provenance for set minuses [17] and aggregate queries [5]. Moreover, Cheney et al. have shown that Why provenance is an instance of How provenance [11].

Where Provenance. In contrast to the previous definitions, Where provenance is defined on attribute level (not

[1]Note, artifacts are not restricted digital items, but can also represent physical objects.
[2]Contains Selections, Projections, Joins, and Unions only.

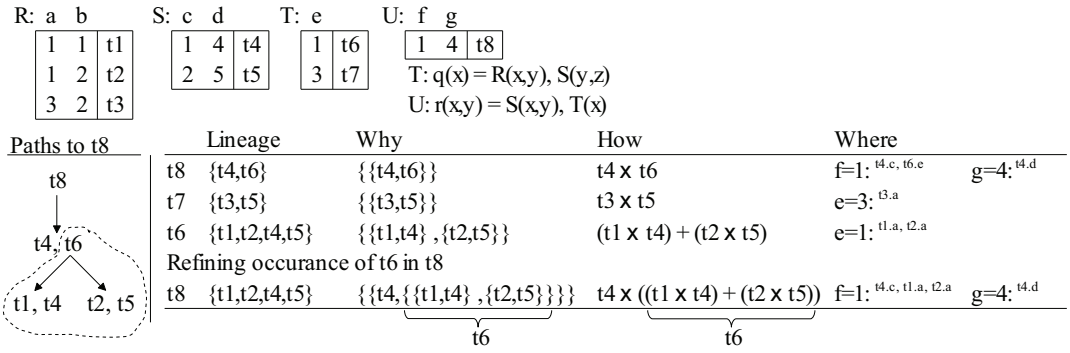


Figure 2: Relationship Between Provenance Terms

on tuple level). By definition, *Where* provenance indicates where an attribute value of an output result was *copied* from. For instance, the Where provenance of attribute f in tuple t_8: $Where(t_8.f) = \{t_4.c, t_6.e\}$ denotes that the attribute in $t_8.f$ was copied from $t_4.c$ or $t_6.e$ (both containing the same value). Unfortunately, it is not possible to use Where provenance for non copy operations (e.g., aggregate functions), which is sometimes possible in How provenance. But it is possible to use Where provenance for operations performing manipulations of the table structure at attribute level (i.e., projection). This is currently not possible in How provenance. Therefore, unifying both approaches is desired. However, Cheney et al. proofed with a counterexample that, according to the currently used formal models, *How* and *Where* provenance cannot be unified in one formal model [11]. On the other hand, they emphasized the similarities between the two notions. For example, the tuples of all attributes in $Where(t_n)$ are present in $How(t_n)$ (but not vice versa).

3. A HIERARCHICAL PROVENANCE FRAMEWORK

In this section, we motivate the necessity of abstraction layers forming our hierarchical provenance framework. Therefore, we introduce a running example that is used for illustrative purposes in the remainder of this paper.

3.1 Provenance Systems

At a very abstract level, systems capturing and evaluating provenance work as depicted in Figure 3. Initially, requirements for the whole system such as granularity, time when data is captured, or availability have to be defined (*preparation*). Afterwards, the system *captures and stores* the provenance data (also known as annotations), which have to fulfill the previously defined requirements. Depending on how the data is stored, the system has to provide a possibility to *query* the provenance data (e.g., by the use of a query language), taking additional issues such as privacy aspects into account [8].

Moreover, the provenance data are *evaluated* with respect to their validity. For instance, developers can use this information to understand how some artifacts have been computed, e.g., how a query result evolved. Beyond that, the system can even reintroduce these results into subsequent computation steps such as computing new query results. However, to validate such results we may face the problem that the necessary provenance information is fragmentary or unreliable. Thus, missing information has to be somehow estimated or validation is simply impossible. In this paper, we address the fragmentation and reliability of existing provenance information. In particular, we show that we can use different layers of abstraction for different purposes based on the fragmentation of the available provenance information. Furthermore, we show that reliability affects all of these layers and thus, can be considered as *cross-cutting* characteristic of provenance.

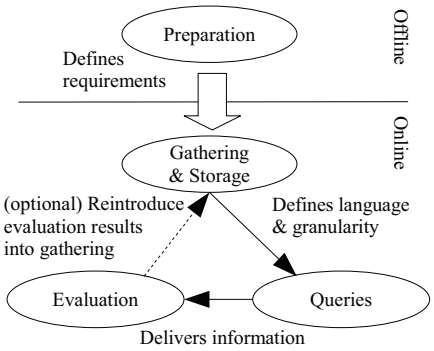


Figure 3: Provenance System

3.2 Necessity for Abstraction Layers

Due to the versatile characteristics of provenance systems, we build our framework on differences regarding uncertainty and fragmentation represented by layers of abstraction. We will motivate the necessity for every layer shortly. Particularly, the single layers form a hierarchical granularity framework for provenance where each layer complements the preceding one in the case that additional, more fine-grained information is available.

For motivating such a hierarchical framework as well as clarifying our intention we introduce a running example. In a current research project[3], we evaluate digital pattern recognition techniques for latent fingerprints based on high-dimensional sensor data. Additionally, we are interested into testing different contactless sensor devices at different materials as well as determining environmental influences. To ensure validity of the evaluation, we have to gather detailed knowledge about the derivation history of all data items originally produced by the sensor(s) and subsequent quality enhancement steps.

[3]Anonymised reference, because of the double blind review.

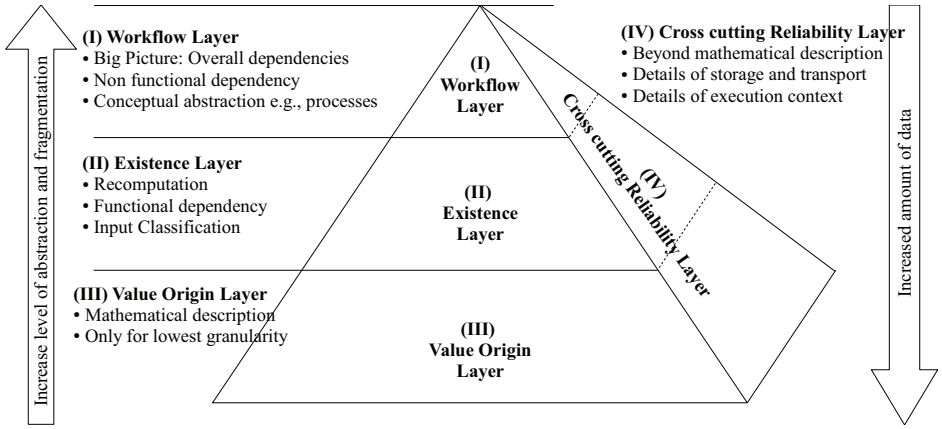


Figure 4: Hierarchical Provenance Framework

For this scenario, we want to be able to use the fingerprints in law enforcement proceedings. To this end, we have to provide detailed and reliable information about the whole processes from scanning the fingerprint to the final results presented at court. However, we have to take into account that the people in court do not have detailed technical background. Consequently, (1) we need a layer that *abstracts* from execution and implementation details. As a result, we can show the complete causal dependencies from input data (the original finger print scan) to the results presented in court and, beyond that, the (possibly certified) processes that created the respective results/data. In contrast, (2) fingerprint experts and software designers require more detailed knowledge about the particular processing steps. Specifically, they have to be able to *recompute* and therefore *validate* foreign results (e.g., when a lawyer doubts evidences). But these people still do not need to know *how* exactly the single results were computed. However, (3) this is important for software engineers implementing the single computation steps. For instance, for quality enhancement of latent fingerprints, we use different filter implementations such as Gabor filter banks. These filters modify the single pixels of the original images (scan data), which, in turn, represent the lowest level of granularity. Consequently, a developer has to know how exactly the filter modifies the pixel values to implement, improve, or choose the right filter. Finally, (4) there are additional factors which we have to take into account. These factors are beyond mathematical descriptions containing details of hardware architecture, storage, and transport. Hence, they complement the information of the previous layers to *improve the reliability* of both, the data itself and the respective provenance information of all preceding layers. We will discuss whether this is a real layer or a different dimension in Section 4.4.

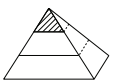
Granularity Layers in the Real World. To avoid overfitting of our hierarchical framework regarding the fingerprint example, we searched the literature for concepts and approaches addressing and generalizing the single layers. Additionally, we took existing approaches as base for each layer and describe the relationships among them. Then, we assembled the results in a hierarchical framework depicted in Figure 4, which is based on the current state of the art.

The top most level contains the Workflow Provenance. Several approaches, such as the OPM [27], basically rely on conceptual abstraction and thus describe only causal dependencies independent of the implementation. In contrast to the first layer, the second requires functional dependencies to recompute and validate results. For this layer we found Lineage [14], Why provenance [9], How provenance [9, 18] and causality [25] as existing approaches. For the third layer, Buneman suggested a model explaining for copy operation, where the single attribute values of an existing tuple stem from [9]. Next, the semirings in How provenance can be used for computing attribute and annotation values for different subsets of database operations. Consequently, this lays the foundation for the third layer of our framework. For the last layer, the basic idea is about increasing (and securing) the reliability of the provenance data of the previous layers such as proposed by Braun et al. [8].
We give a detailed and comprehensive overview for all layers and their relation to each other in the next section.

4. THE FOUR LAYERS

In this section, we introduce the four abstraction layers and their relationships to each other. Additionally, we point out their usage and possible refinements for each layer.

4.1 Workflow Layer



This layer is a conceptual abstraction, representing what happens at coarse grained level totally independent of implementation (if there is one). Hence, this layer shows causal dependencies while hiding details about functionality. Moreover, this is a starting point for validating, comparing, or discussing results. Finally, this layer can be used for automatically validation of derivation histories which might be produced in distributed systems not having access to each other.

4.1.1 Extended Open Provenance Model

In the workflow layer, we rely on the OPM [27], that was developed to be a pure conceptual abstraction of provenance information. We rely on the OPM, because it is a consensus explicitly designed for workflow provenance. Moreover, most of literature addressing workflow provenance such as [1, 28, 30] refers to a directed acyclic graph and is therefore similar to

the OPM. To bridge the gap between coarse grained workflow provenance and fine grained forms of provenance [12], we present an extension of the OPM in this section. Finally, we explain usage and limitations of this layer.

As already mentioned in Section 2.1, we use additional artifact types to have more semantics in the model. Within our extended OPM, an artifact has (1) a name, a (2) role, (3) a type showing whether this is a complex or a simple artifact and furthermore a unique id as well as a mark denoting if this is an initial or result artifact (cf. Figure 5). For initial artifacts, we have *no* knowledge about previous causal dependencies, but about subsequent ones. Hence, backtracking of one of these artifacts is impossible. In contrast, for result artifacts we have knowledge about previous causal dependencies, but not about subsequent ones due to the fragmentation of provenance. Thus, these are the last artifacts for which we can form the dependency graph. Similar to artifacts, each process has (1) a name, (2) a hierarchy label, and (3) a type.

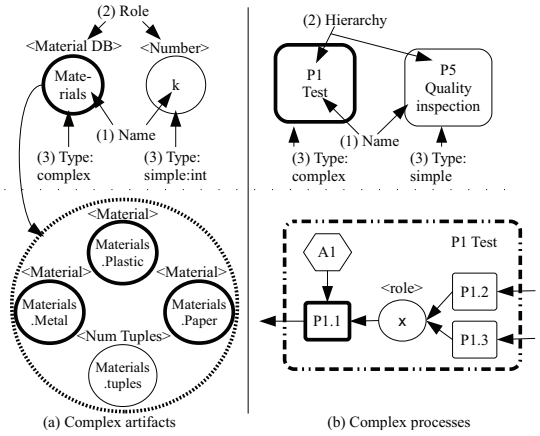


Figure 5: Complex Artifacts and Processes

4.1.2 Stepwise Refinement

Regarding our extended OPM, the most important extension is the introduction of complex artifacts and complex processes allowing stepwise refinement of granularity. We argue that this is the key issue to bridge the semantic gap between coarse grained workflow provenance and different fine grained provenance models (accumulated in the subsequent layers). In fact, it is possible to refine the processes of the *workflow* layer until reaching the connection points to the *existence* layer. In the following, we will introduce our extension of the OPM and how this allows for stepwise refinement.

Complex Artifacts. In Figure 5 a, we depict the concept of stepwise refinement used for complex artifacts. A complex artifact such as the *Material DB* contains a certain number of either complex or simple artifacts. This forms a *tree structure*, where the root of the tree is the top most complex artifact (in this case the *Material DB*), inner nodes are again complex artifacts and the leafs are simple artifacts, directly linked to a primitive value (e.g., an integer). The position in the tree is inherited in the name of node. For instance, the number of tuples in the *Material DB* is referenced as *Materials.tuples*. This is similar to object-oriented programming where this notion is used to reference fields or methods. A complex artifact inherits its data such as tuples of a DB table or members of an object. Furthermore, such artifacts inherit *additional* annotations that contain meta data about the artifact itself. Note that due to the fragmentation of provenance or due to privacy issues [8] not all data may be available (i.e., several nodes might be missing).

Complex Processes. Analogously to complex artifacts, we also introduce complex processes for stepwise refinement (cf. Figure 5 b). The refinement of a process is an acyclic graph showing the causal dependencies of the parent process in more detail. To reference the parent process, the process contains a hierarchy label. For instance, process P1.1 is within the refinement graph of P1. Basically, the process has to collect provenance data on its own or may use integrated monitors [23]. Moreover, this data is always collected at the lowest level of granularity, that is, the leaf level of the tree structure. Consequently, all upper levels are aggregations of this most detailed level.

Implicit Dependencies. To allow processes the usage of artifacts with different levels of granularity in one graph (i.e., only some artifacts are refined), we introduce the concept of *implicit dependencies*. For instance in Figure 6 a, *Process 1.2* uses artifact *a.2.2*. As a result of this *explicit* dependency, there are additional implicit *use dependencies* from P1.2 to every ancestor node in the *Artifact Tree of a*. Hence, P1.2 has implicit a *use dependency* with *a.2* and *a*. Moreover, we add additional *use dependencies* from *a.2.2* to every parent process of *P1.2*. Thus, a.2.2 has a *use dependency* with *P1*. Finally, this is repeated for every pair of parent artifact and parent process recursively, up to the lowest granularity level. Therefore, *a* and *P*1 have a *use dependency* too.

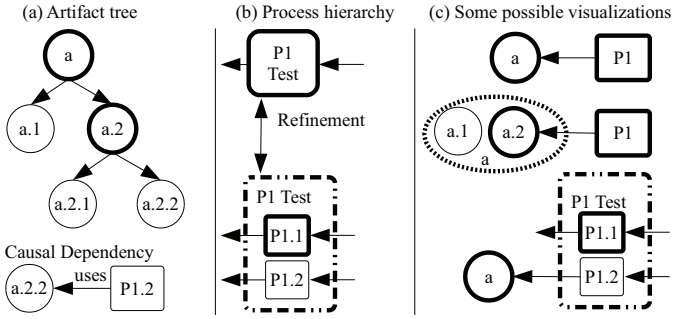


Figure 6: Visualization of Implicit Dependencies

4.1.3 Usage and Limitations

To explain the usage and limitations of this layer, we refer to the motivating example from Section 3.2. In Figure 7[4], we depict the causal graph for two result artifacts, namely quality assessment and age interval of a fingerprint. Both artifacts are final results, and thus we have no knowledge about any subsequent causal dependencies. Hence, we cannot build any graph using these artifacts as input. However, the graph reveals *where* the initial finger print was taken from and *how* the results have been created at a very coarse-grained level hiding lots of possibly important details.

[4]Note that we omitted labeling the causal dependencies for clarity reasons. To avoid ambiguous dependencies, we use only *use* (from Process to Artifact), *wasProducedBy* (Artifact to Process), and *wasTriggeredBy* (Process or Agent to Process) in this example.

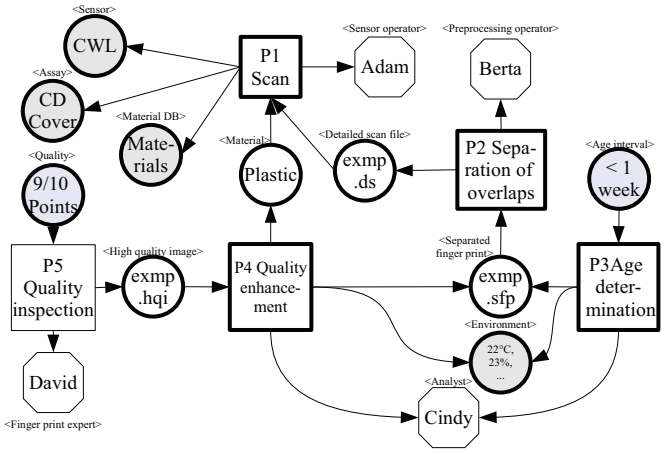


Figure 7: Provenance Graph containing Complex Artifacts and Processes

Backtracking. In the example, the provenance graph exhibits its maximum level of backtracking. This means that we tracked back *all* dependencies to initial artifacts, where backtracking is defined as follows: First, the level of backtracking for each vertex in the graph used as starting point is defined as *Level 0*. When increasing the backtracking level, we expand the graph for each causal dependency until we reach some process. At the same time, we add all artifacts and agents having a direct dependency with this process to the backtracking level. In Figure 8, we depict the first three backtracking levels for the *quality artifact* of the example. We argue that stepwise backtracking is important to simplify understanding of these graphs because provenance graphs can be very large and therefore hard to understand.

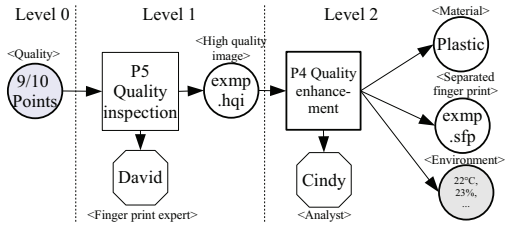


Figure 8: Exemplary Backtracking Levels

Graph Merging. Unfortunately, by simple backtracking it is *impossible* to create the graph in Figure 7. This is the case because both result artifacts (quality and age interval) use the *separated fingerprint* artifact as input. Consequently, there is no path connecting both result artifacts in one graph. As a result, by backtracking both artifacts of our example, we obtain two provenance graphs that can be merged when backtracking reaches *Level 2* for the *quality artifact* and the *age interval* artifact. Thus, merging graphs is important to a) show whether some artifacts share a common derivation history (i.e., share a subgraph) and b) reconstruct graphs based on the derivation history of artifacts.

Granularity Refinement. Except for process *P5 Quality Inspection*, all processes in our example in Figure 7 are of type *complex* and thus can be refined. For instance, in Figure 9 *P1 Scan* is refined, that is, the complex process from Figure 7 is replaced by its corresponding subgraph. Due to the concept of implicit dependencies, we can have both, complex and refined processes, in one provenance graph and thus show details that are not available on the coarse-grained level of granularity. For instance, our refined process in Figure 9 reveals that the whole scanning process performs two scans. First, *Scanner Operator Adam* triggered a coarse scan (P1.1) that was used to find *regions of interest* possibly containing a fingerprint pattern. Second, the Scanner Operator triggered a detailed scan (P1.4) using the information obtained from the coarse scan (P1.1). Moreover, the graph depicts that P1.1 triggers P1.2 and P1.3 automatically.

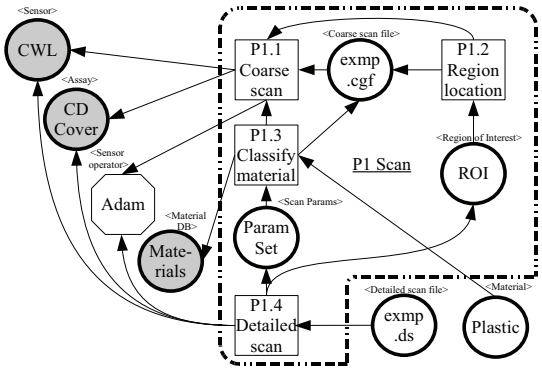


Figure 9: Refinement of the Scan Process P1

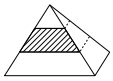
Automatic Validation. The provenance graph can be used for automatic validation. To this end, we have to specify for each process a list for certain valid combinations of input roles and resulting output roles (or a negative list respectively). Moreover, it is possible to define certain constraints such as the number of output artifacts has be greater than the number of input artifacts etc. Consequently, a monitoring program having access to the provenance data and to these constraints can detect anomalies.

Limitations. In contrast to the next layer, we do not restrict processes to functions. This means that providing the same input *does not necessarily* have to result in the same output[5]. Reasons therefore are missing input parameters such as configuration parameters or class members. For instance, in P5 *Quality Inspection* a fingerprint expert evaluates the quality of the fingerprint image. Although we assume that he is an expert, different environmental factors such as different light settings in different laboratories might change the quality estimation result especially for borderline images. Consequently, to reproduce results we have to assume functional dependencies that we cannot express with processes as conceptual abstraction. Furthermore, in some cases, such as manual quality inspection, it may be impossible to guarantee functional dependencies. Consequently, we argue that approximating the functional dependency with refinements is practical and sufficient and better than having no provenance at all. To determine the similarity of artifacts of the same role we suggest the use of distance metrics as applied in [15].

[5]Here, identical artifacts means that the artifacts trees have the same structure and the primitive value(s) within an artifact contain the same value(s), excluding annotations.

Related Approaches. Biton et al. use a similar concept to create *user views for arbitrary scientific workflows*. First proposed in [6] and then improved to avoid to induce loops in the workflow [7], it allows to merge composite modules (similar to our complex processes) to reduce the amount of provenance data presented to a user. As this model was developed to visualize and query previously captured provenance data, not to determine what provenance data to capture, the focus is slightly different. However, the similarity is in the way how to link different levels of granularity. In contrast to our approach, Biton et al. only refine processes (not artifacts) resulting in a complex merge procedure, which possibly forbids merging several processes. A solution therefore are our implicit dependencies allowing us to more flexibly refine processes if only parts of an artifact (e.g., a tuple in a table) is used. Moreover, to build the provenance graph, we use an inverse temporal order (from output to input, not vice versa), which is a tribute to the fragmentary nature of provenance and the application scenario in that we want to determine the past derivation history and not the future use. In fact, a result cannot know what it is used for in future, but it may know which artifacts (or respective subsets) where used to create the artifact itself. As a result our approach is more flexible, as we do not assume that in the graph there is one (complete) input node, but for instance also accepts hidden inputs such as state variables from a previous run having a certain impact on the current result building process. Finally our artifact trees may also contain the primitive values (if known) and are thus not restricted to simple id's written in the log files used to create the graph in [7].

4.2 Existence Layer



While the previous layer addressed causal dependencies, this layer focuses on result validation. Therefore, we only consider artifacts a computation step created and thus *exist*. In turn, we do not consider artifacts that could have been created potentially, relevant for instance for query non answers. To allow result verification, we assume a *functional dependency* for artifact creation: $(f : a^n \rightarrow a^m)$. Consequently, we can recompute and thus validate results. Moreover, depending on the available information regarding the behavior of the single functions, we split the existence layer into three sub levels. In the following we explain the three sub levels, point out how refinement takes place in this layer and its limitations.

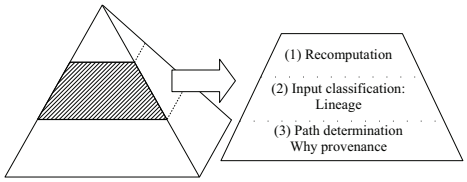


Figure 10: Sub Levels of the Existence Layer

4.2.1 Sub Levels of the Existence Layer

Subsequently, we explain the purpose, use and relationship between the three sub levels of this layer.

(1) Recomputation. The first sub level allows recomputation and thus validation of results. To recompute the result(s), we have to ensure that the input of the single functions in this layer is identical, which is not always a trivial task. Currently, most authors assume that we know the input. For example, in DB formal approaches such as How [18] and Why provenance [9], the authors implicitly assume that all input (relations) are known. In the relational model, knowing the query also means knowing the input, because all input relations are part of the `FROM` clause(s). But in different data models or programming paradigms this is not as simple. While in functional programming the input are the arguments in the function call, in object-oriented programming we additionally have to take into account class members and different static variables. Consequently, knowing the input is the minimum requirement for recomputation.

(2) Classifying the Input. Splitting the input into possibly necessary (*endogenous*) input and never necessary (*exogenous*) tuples can improve the understanding of what actually happened during computation. Again, we start with DB formalisms and explain which of these definitions can be adapted to different data models and programming paradigms. According to Meliou et al. [25, 26], a set of input tuples I for some query $R = q(I)$ producing an output tuple t_i consists of endogenous tuples I^e (tuples used in at least one path) and exogenous tuples I^x (not used in any of the paths). Both, I^e and I^x are subsets of I, their intersection $I^e \cap I^x = \emptyset$ is empty, and their union restores $I = I^e \cup I^x$. Moreover, the authors differentiate endogenous tuples into counterfactual and actual tuples due to the set semantics that can lead to multiple paths producing one output tuple.

Consider the following example: $\{a\} = q(x) = R(x), S(x, z)$ over an instance (I): $R\{a, b\}, S\{(a, b), (a, c)\}$. Obviously, the tuple $R(b)$ forms I^x because there is no join partner in S. But there are two ways to compute the query result: (A) Join $R(a)$ with $S(a, b)$ or (B) join $R(a)$ with $S(a, c)$. Therefore, R(a) is a counterfactual tuple $t \in I^e_c$ because removing R(a) from I would remove $\{a\}$ from the query result. Generally, Meliou et al. define a counterfactual tuple t for some result tuple r w.r.t. to some query q as $r \notin q(I^e - \{t\})$ if $r \in q(I^e), t \in I^e_c$ holds. In contrast, we can remove either $S(a, b)$ or $S(a, c)$ from I^e because there is still one path creating the same result. Consequently, these tuples are actual tuples because they can be part of a certain subset $(\Gamma \subseteq I^e)$ which can be removed from the input still producing the tuple r w.r.t to query q. Note, that Meliou et al. relate *causality* based on the definition of Halpern and Pearl [19] to *How provenance*. We stick to their results. But according to our framework, causality is finding all possible inputs which might have produced t_n w.r.t. q. Therefore, causality is important if we know only parts of the input and want to compute (all) possible inputs or query non-answers. By contrast, in linage we assume we know the input and want to validate what happened. Consequently, $\forall t_i \in q(I)$ I^e is equivalent to $Lin(q, I, t_i)$ and $I^x = I - Lin(q, I, t_i)$.

When adopting these definitions to different data models without set semantics, we can omit the differentiation between actual and counterfactual input artifacts, because all actual inputs are also counterfactual. This simplifies the classification of input determined at the previous sub level, because we can simply omit this differentiation. Unfortunately, currently there are no formalisms automatically computing the input classification for different data models, such as they exist for the relational data model. Consequently, we have

to manually collect these annotations in the program itself.

Nevertheless, this sub level improves understanding the relation between input data and result computation (query equivalence) as follows:

- Every input artifact in I_c^e *cannot* be removed without changing the result of the operation.
- Every input artifact in I^x *can* be removed without changing the result of the operation.
- (Only with set semantics) An input artifact $\in I^e$ but $\notin I_c^e$ can be removed from I without changing the result, because there is at least one path not using this tuple to produce the same result.

(3) Determining Paths. As we stated in Section 2.2, a *path* is an acyclic graph showing the sequence of operation (vertices) and respective connections (edges) to input artifacts. In set semantics, multiple paths may lead to *one* result and thus paths are highly related to minimal witness bases [9, 25]. Furthermore, formalisms such as Why and How provenance denote the computation paths. The main difference to the previous sub level is, that an artifact may occur at different nodes in the graph (see Figure 2).

In summary, the former two sub levels are approximation of the path determination level. We need these approximations because of fragmentary knowledge about implementation details. For example, in API programming or for aggregate functions in databases, we only know the function name and the arguments to supply. Therefore, we have to assume that all of the input artifacts are counterfactual ($I = I_c^e$) and consequently no exogenous artifacts exist. This allows to recompute and consequently validate results. In the second sublevel, we classify the input (if possible due to our knowledge) to know which input artifacts (I^x) can be omitted without changing the result. Furthermore, we determine the existence of multiple paths ($I^e - I_c^e \neq \{\emptyset\}$), artifacts contained in every path (I_c^e) and artifacts which might be part of a certain contingency ($I^e - I_c^e$). However, in the third sublevel, we determine paths itselves. Each path p in the set of paths P ($p \in P$) w.r.t. a specific operation (e.g., query) o and Input I creating result r contains all counterfactual artifacts in I_c^e and (a possibly empty) subset of non-counterfactual artifacts $I_{c^{-1}}^e = I^e - I_c^e$. Consequently, the following equations hold: The union of all artifact sets of all paths is equivalent to the set of endogenous input artifacts $\bigcup_{\forall a \in p \in P} = I^e$. Moreover, the intersection of all artifact sets of a path is equivalent to the set of counterfactual artifacts $\bigcap_{\forall a \in p \in P} = I_c^e$. Finally, the set minus of all non counterfactual artifacts $I_{c^{-1}}^e$ and a specific path produces a certain contingency $\Gamma = I_{c^{-1}}^e - p$. Obviously, a contingency is a set of endogenous artifacts that we can remove from I^e so that exactly one path remains for creating the result r.

4.2.2 Refinements

At the existence layer, there are two possible refinements at each sub level showing different results: (A) Backtracking of artifacts and (B) Implementation refinement. The limitations of both refinements are highly related to the fragmentation of our knowledge as we will explain in the following.

From Results to Initial Artifacts. For each of the sub levels it is possible to *backtrack* artifact occurrences in the same way as with the previous layer (cf. Section 4.1.2). The backtracking stops when reaching an initial artifact, which is the first element we have provenance information for. In a nutshell, backtracking operations can be summarized as follows: Union of the whole input sets for sub level one, union of endogenous input sets I^e in sub level two, and merging of directed acyclic graphs producing a new directed acyclic graph in the third sub level.

Table 1: Refinement with Fragmentary Knowledge

Fragmentary Knowledge	SubLevel 1	SubLevel 2	SubLevel 3
1 r=foo(a,b,c,d);	I={a,b,c,d}	I={a,b,c,d}	r ↓ a,b,c,d ⌋ foo
1 int foo(a,b,c,d){ 2 ret=foo2(a,b); 3 ret+=c; 4 return ret;}	I={a,b,c,d}	I^e={a,b,c} I^x={d}	r ↓ ret, c ↓ foo2⌈a,b ⌋ foo
1 int foo(a,b,c,d){ 2 ret=min(a,b); 3 ret+=c; 4 return ret;}	I={a,b,c,d}	(I^e={a,c}, I^x={b,d}) or (I^e={b,c}, I^x={a,d})	r ↓ ret, c or r ↓ ret, c ↓ a⌋ min ⌈b ⌋ foo

Refining Implementation Details. This kind of refinement is showing more implementation details such as additional knowledge of the internal structure of functions. For instance, a function can call several function (e.g., `foo2` and `min()` in Table 1) and operations such as +,-, etc. To explain this refinement and the influence of fragmentary knowledge, we refer to the example depicted in Table 1. In this example, there is a function `foo()` taking four arguments as Input I (for simplicity we omit argument types) and returning one (result) artifact. In the first part of the example, we assume that we do not have detailed knowledge about this function. Hence, we cannot classify the input leading to one path, assuming that all of the input is counterfactual. In the second part of the example, we assume that we have additional knowledge about the internal structure of `foo()`. Note that we still have fragmentary knowledge, because we do not know what `foo2()` in Line 2 calculates. Hence, it is possible to *refine* implementation details by revealing the sequence of functions and inputs in the example. To keep the reference to the containing functions such as `foo()`, (sub) paths (i.e., the vertices) are annotated with the function they belong to. The refinement stops when reaching operations only or because of fragmentary knowledge about functions. In the example, we do not know what `foo2()` actually does and thus we have to approximate I^e from $I = \{a, b\}$ (i.e., we do not know whether this function actually uses the arguments). This introduces uncertainty into our provenance information. However, with additional knowledge about `foo2`, we can calculate I^e. For instance, if it works like a min function (i.e., is equivalent to an operation), we cannot further refine the implementation, but it is possible to determine the path according to the input I.
The advantage in databases is that, for monotone queries, we have formalisms that automatically and therefore without uncertainty compute:

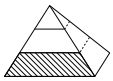
- For sub level 2, the set of necessary input tuples I^e: Lineage, Why provenance and How provenance,
- For sub level 3, the set of all paths: Why provenance and How provenance.

Link to Parent Layer. As recently introduced, a function that is called within the body of another function such as `foo2()` from `foo()` has an annotation to keep the reference to the calling function. In contrast, the function `foo()` in Table 1 is the top most function and thus does not contain a link to a calling function. However, such functions have a link to the abstract process in the preceding layer showing the respective part of its computation for a particular artifact. Moreover, these functions may also consume artifacts from the workflow layer.

The combination of both, refinements and the link to the workflow layer, allows us to increase our understanding of what actually happens. Note that especially the refinements require proper tool support to visualize certain excerpts from the derivation history.

Limitation of this Layer. In the example in Table 1, we did not consider whether the arguments supplied to `foo()` are primitive numeric values or complex objects. However, at this layer we do not care *how* the primitive values within an artifact or its annotations (if it is complex) are calculated. This is restricted to the semantics of the functions and operations denoting to what extent the primitive values of artifacts contributed to a result. This is in the scope of the next layer.

4.3 Value Origin Layer



With the previous layer, we focused on the determination of paths responsible for the existence of artifacts. In contrast, this layer is about the origin of the primitive values within existing artifacts such as attribute values in the relational data model. In the following, we explain how the origin of values can be addressed by existing approaches and application for different data models and the relationship to the *existence* layer.

4.3.1 Structure of Artifacts

As mentioned in Section 4.1.2, we differentiate between complex and simple artifacts. Simple artifacts are directly related to one primitive value such as an *int* or *char* type. In contrast, complex artifacts may contain several complex and simple artifacts forming a tree structure, where the leaf nodes are always simple artifacts. Consequently, each artifact is a container for primitive values at the lowest level of granularity. For instance, consider a relation containing *regions of interest* (ROI) from our fingerprint example. By our means, a ROI is a rectangular part of a coarse scan probably containing a fingerprint. This ROI is input for the detailed scan process. It is used to scan the physical object again with higher resolution showing details such as sweat pores that are not available in the coarse scan. To increase productivity in our illustrative example, we want to scan in detail those physical objects first carrying the greatest amount of fingerprints. Therefore, we store additional annotations.
In Table 2, we show the relation storing the ROIs permanently. Each tuple has a surrogate primary key from some sequence within the DBMS, two points denoting the left upper and right lower bounds of the rectangle, and a foreign key referencing the coarse scan file of the physical object. Moreover, there are general additional annotations for every artifact containing the unique artifact ID, name, role, type, and mark (Section 4.1.2). Finally, for each role there are role specific annotations (cf. Table 2 and Figure 11).

Table 2: ROI Relation

ROI	PK: int	l_u: Point		r_B: Point		FK: int		Role specific annotations	
		x: int	y:int	x: int	y: int			Probability	ROI Locator
t1	1	12	49	287	413	5	→	0.85	v.0.7.3a
t2	2	12543	736	12781	1052	5	→	0.91	v.0.7.3a

4.3.2 Value Origin with Existing Approaches

Based on the structure of artifacts, we are now interested how we can determine the value origin of certain artifacts. The value origin of some simple artifact A w.r.t. an operation O for a path P shows *how* a particular subset of the primitive values within the artifacts contribute to A. For instance, consider the query Q to get all ROIs in Figure 11. Furthermore, assume for simplicity that there are only two ROIs for scan five. For normal computation, the result would look as follows: $t3 : \{(5, 2)\} = Q(ROI))$. In contrast, when including provenance in this scenario, the results looks as depicted in Figure 11. Meanwhile, recapitulate that we want to scan those objects first carrying the greatest amount of fingerprints to increase productivity. Thus, we additionally store the probability that there is at least one finger print on the physical object and the ROI Locator Version. Now, we are interested into approaches explaining the value origin of the query result of Q including the respective annotations.

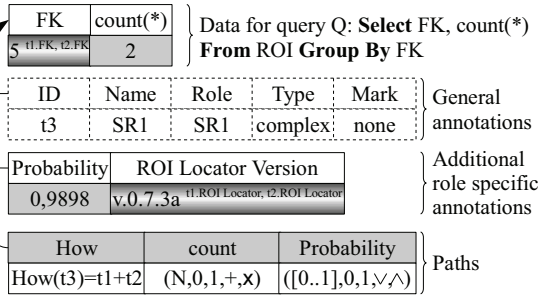


Figure 11: Computation with provenance

Where provenance. In databases, the *Where provenance* formalism is able to determine the origin for pure copy operations only [9]. By definition, this works for the data such as the attribute FK, because this is what it was designed for. In contrast, when using Where provenance for annotations we face the problem that there may be competitive annotations. For instance, imagine that the first ROI was computed with ROI Locator Version v.0.7.3a while the second was determined by v.0.6.8. In this case, we currently add the symbol a showing that the version is ambiguous (which must not happen). Because this is a domain-specific solution, we consider different solutions allowing annotations to collect sets (both versions) or rules to determine the dominant annotations (e.g., the later version if there are no functional changes, but only an increase of performance).

Semiring Model and Extensions. In contrast to Where provenance, we can use the semiring model to calculate the

results of queries (e.g., for attribute or annotation values). In the original version this works for SPJU queries [18] and has been extended for set minuses [17]. Finally, Amsterdamer et al. extended the model for aggregation support including optional group by operations with some limitations [5]. The general challenge in this model is finding the semiring with the right semantics as in the probability example [11]. Moreover, it is currently not possible to determine the origin of the PK in our example, because it is linked to a sequence. To determine the origin, the model has to have access either to the sequence or to the previously created tuple (requiring an order). Finally, to the best of our knowledge it is not possible to use constants.

Usage for Different Data Models. While *Where provenance* and the original semiring model are limited to databases, Amsterdamer et al. emphasize that the extension for aggregate queries is independent of the data model and therefore might be used to capture *automatically* provenance information for aggregate functions. However, the concepts behind Where and How provenance, are also suitable for different data models. But for now, the respective capturing of value origin in different data models has to be done manually or is not performed at all.

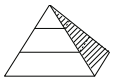
4.3.3 Relationship to the Previous Layer

Independent of the way how we determine the value origin, there are several relationships to the previous layer. First, for each primitive artifact in every path there is one mathematical description. In *Where* provenance these are the locations the primitive values are copied from. In the semiring model this is a series of operations w.r.t. to a particular semiring. Second, all simple artifacts used in one mathematical description are part of the corresponding path [11].

Backtracking Refinement. In contrast to all previous layers, there is only one possibility for refinements, because the value origin is always defined on the lowest level of granularity. Consequently, there is no granularity refinement. However, backtracking is still possible by means of replacing a simple artifact in the mathematical description. For instance, in the initial example in Figure 2, in Where provenance of attribute $t8.f$ $Where(t8.f)$ are the locations $t4.c, t6.e$. Now we can replace the location $t6.e$ with $Where(t6.e) = \{t1.a, t2.a\}$ (i.e., remove $t6.e$ and add the result $\{t1.a, t2.a\}$). Limitations of backtracking are related to fragmentation. In contrast to the previous layer, we require detailed knowledge about how exactly the simple artifacts contribute to the result of the computation and cannot estimate this information. Consequently, further backtracking is not possible if we do not have this knowledge for an operation in the path.

Limitations of this Layer. As this layer works on mathematical description, it reaches its limitations for instance when details of the (hardware) dependent execution context (e.g., for timing experiments) are required or even when trying to prevent or detect possible attacks.

4.4 Cross Cutting Reliability Layer



As all previous layers are dealing with granularity and fragmentation, this layer addresses the reliability of artifacts itself and respective provenance data. Therefore, this layer decreases the degree of *uncertainty*. Because this is possible for all previous abstraction layers, we call this layer *cross cutting*. Particularly, this looks like a different dimension to us. But this dimension is not totally orthogonal, as fragmentation may cause uncertainty as well (see Section 4.2.2). Hence, we keep calling this dimension of provenance a *layer*.
The purpose of all preceding layers was to improve our understanding of what happened (e.g., for validating results) and are restricted to our fragmentary knowledge. Unfortunately, this information can be wrong for arbitrary reasons. For instance, a malicious attacker could have changed the data itself or corresponding annotations. Moreover, there can simply be errors when collecting the information. This is especially the case when there is no possibility of capturing the information automatically (e.g., by a formalism). Consequently, this layer addresses reliability and trust in provenance information [22, 23, 24].

4.4.1 Challenge: Increasing Amount of Data

Consider the example from Table 2. To be really sure that all ROIs are calculated with the same ROI Locator version, we furthermore annotate the version annotation with a security hash sum of the version binary. Moreover, to ensure that this artifact remains unchanged, we have to add an additional signature computed over the the whole artifact (including annotations).

The Crosscutting Characteristic. In the recent example, we included a signature to ensure that a specific artifact remains unchanged. Actually, we want to do that for all provenance data collected in the preceding layers such as the graphs in the Workflow Layer or the paths in Existence Layer. Moreover, we may want to propagate that we secured the execution context for instance with TPM modules as suggested in [23]. But this is not only related to security issues. For instance, for some application scenarios we need a result within a certain amount of time. Therefore, we perform several timing experiments using different algorithms for computation of ROIs. To compare these results, we need to know about the hardware environment in case that the tests are not performed on the same computer. As a result, the amount of data to store increases rapidly, which is also denoted by area of the pyramid slice in the framework visualization (Figure 4).

Annotating Annotations. An interesting problem is that annotating annotations furthermore increases the amount of data. As mentioned previously, we want to include additional annotation to increase the reliability of annotations or values. Suppose we want also know which hashing algorithm was used, in which implementation, and how these additional annotations have been collected. The problem is to determine when to stop annotating, because when doing this for every piece of data (i.e., every artifact, annotation) the amount of annotation data is multiplicity of the values within the simple artifacts (e.g., in Figure 11) [11].

4.4.2 Current Research on Reliable Provenance

Although reliability is an increasingly interesting topic, only minor work exists that addresses this layer. For instance, McDaniel et al. tackle the problem where to place the software that collects provenance information (monitors). For non-formal approaches, they emphasize the need for securely

deploying provenance especially in distributed systems [24]. Similarly, Tan and Lu argue that there are special problems in SOA [31] or cloud [22] environments. Consequently, the required mechanisms for reliable provenance depend on architecture and use case. For instance, Lyle et al. point out that it is applicable to protect the computation environment (i.e., processes or functions) and monitors with hardware-based methods, such as TPMs, when malicious effects (e.g., from the user) have to be considered [23]. Moreover, Schäler et al. suggested to save reliable provenance data within multimedia data itself using invertible watermarks [29].

Basically, we identified two important points to take care of: (1) The execution context of processes or functions (e.g., usage of TPMs, hardware architecture, etc.) and monitors as well as (2) details about storing and transportation of artifacts including the detection of (malicious) modifications.

5. CLASSIFICATION OF APPROACHES

According to our literature research, there are three different approaches for collecting provenance. First, there are domain-specific approaches with a restricted set of functions. Second, formalisms capture the desired information automatically and totally transparent on the operator level. Finally, when there are no formalism and a limited function set is not sufficient, the desired information has to be collected manually. Indeed, this is the state of the art for many data models and programming paradigms. In the following, we give examples for each approach and explain how these concepts or approaches interact with our hierarchical framework. Furthermore, we point out how the single approaches are related to uncertainty and fragmentation.

5.1 Domain-Specific Approaches

Provenance applications such as Kepler [3, 28], Karma2 [30], or Panda [21] represent a domain-specific all-in-one solution for designing the workflow as well as organizing the collection and storage of provenance data. Although these solutions are theoretically able to collect data of the first three layers and capture provenance automatically, they face two major problems. First, the granularity of the collected provenance data is limited according to the granularity of usable functions. In fact, step-wise refinement is not possible, which is a key issue to link coarse grained forms of provenance to more fine grained ones (see Section 4.1). Second, because of limited available functions and their composability, these solutions are limited w.r.t. to generality and explicitly designed for their domain-specific use case. To overcome this limitation, these approaches may offer a possibility to extend their function set with user-defined functions. In this case, the solution cannot automatically determine the endogenous input of theses functions (Layer 2, sublevel 2) which seems to be a vicious circle. Finally, to the best of our knowledge the treatment of secure storage and transport of (provenance) data is currently not in the scope of theses solutions.

Interestingly, *API-calls* and *spreadsheet programs* are very similar to the previously mentioned solutions, because they use a limited set of function hiding the specific implementation. Consequently, we cannot determine the endogenous input or use refinements. Moreover, there is no link to an abstract process of the *workflow layer*. Hence, these concepts are limited to the input determination of the existence layer.

5.2 Formalisms for Provenance Capturing

Formal approaches, mainly for databases, capture fine-grained provenance and thus are restricted to the Existence (e.g., Lineage [14] and Why Provenance [9]) and Value Origin Layer (e.g., How Provenance [18]). Because these formalism directly work on the operations (e.g., projection in databases), their results are very reliable (Layer 4 execution context). Hence, domain-specific solutions can increase their reliability when (partly) sticking to these formalisms (e.g., the TRIO System [2] captures a form of Why and Where Provenance [11] as well as ORCHESTRA [16]). This is important when extending domain-specific solutions with own functions breaking the vicious circle.

Recent approaches such as the extension of the semiring model for aggregate queries seem promising in this area [5]. In fact, in a very recent publication [4] Amsterdamer et al. use a variant of the semiring model to link coarse and fine-grained forms of provenance in a framework highly related to the fragmentation dimension of this work. They map all expressions of the Pig Latin language (except updates) to bag-semantic nested relational algebra operations, to make them applicable for a variant of the semiring model. As a result, their approach has a *strong formal foundation*, allowing for instance deletion propagation (What if?). Furthermore, they are also able to zoom in and out for processes (modules), but not for artifacts as we can and they as well differentiate between initial and final artifacts. However, their framework is designed to work in a closed system and is therefore less flexible as it is limited to the operations of that language and has preconditions that we argue cannot be ensured generally. For example their workflow provenance is equivalent to our Recomputation Entry (Layer 2 Sub-level 1), that means that non-functional dependencies, for instance for non-computation steps are not considered at all, as they presume to have knowledge about all input artifacts (including artifacts describing the current state of a process). As a result, their framework only addresses Layer 2 and 3. Additionally, they presume complete control of the whole provenance system, which is possible as they rely on their closed system (allowing for instance to use globally unique identifiers). Hence, they do not consider system border crossings of an heterogeneous IT landscape. Finally, they do not use our fine-grained levels in the Existence Layer, allowing us to react more flexible because of missing fragments caused for example by fragmentary knowledge or privacy requirements.

5.3 Manual Collection

Manual collection of provenance data is always required if there is no domain-specific solution or formal approach. For instance, when manually annotating data or writing program code, this currently is the only possibility to capture provenance. Consequently, this way is the most flexible one. Hence, it is possible to collect data for all levels of our provenance framework. Unfortunately, for the same reason this approach is the most unreliable one. As the collection is manual, we have to consider collection errors (e.g., wrong manual annotation or programming errors within the monitor) as well as possibly (malicious) modification of the collected provenance data.

Table 3: Possibility of Approaches to Address the Fragmentation and Reliability Layer

(a) Addressing the Fragmentation Layers | (b) Possibility to Address Reliability Layer & Automatism

	Layer 1	Layer 2 - Existence			Layer 3	Layer 4 - Reliability			Automatism
	Workflows	Input	Endogenous input	Paths	Value origin	Execution context	Storage & Transport	Inherent reliability of captured prov.	Automatic capturing
Domain Specific App.									
- General	Y*	Y	Y*	Y*	Y*	Y*	-	Y	Y
- Extensions	-	Y	-	-	-	-	-	Y	Y
- API / Spreadsheet	-	Y	-	-	-	-	-	Y	Y
DB Formalisms									
- Lineage	-	Y	Y	-	-	Y	-	Y	Y
- Why Prov.	-	Y	Y	Y	-	Y	-	Y	Y
- How Prov.	-	Y	Y	Y	Y	Y	-	Y	Y
- Where	-	-	-	-	Y	Y	-	Y	Y
Manual Collection	Y	Y	Y	Y	Y	Y	Y	-	-

Legend: Y → Yes, - → No, * What data is actually captured depends on the implementation.

5.4 Qualitative Evaluation

In Table 3, we summarize the information presented in this section. In part (a) of this table, we present to what extent an approach is able to capture the data that correspond to layers of our hierarchical framework. Note that this does not mean it captures all of the data in this layer. For instance, domain-specific solutions have a limited function set and database formalisms are mostly restricted to the relational model (i.e., operations like projection). However, recent work concerning provenance for aggregate queries may be able to overcome this limitation [5]. In Table 3 b, we denote whether an approach addresses *layer four* of our framework and whether this approach captures the provenance information automatically. For all formalisms, Layer 4 storage and transport is out of scope. Hence, they only address the execution context aspect of this layer. Similarly, the addressing of this layer in domain-specific solutions highly depends on its implementation. Assuming the implementation of the solution, API, etc. is correct, there is also an inherent reliability for the execution context. Note that inherent reliability is not able to prevent malicious modification of a (skilled) attacker. By contrast to all preceding approaches, manual collection is able to explicitly address layer four (e.g., by signatures, usage of TPMs etc.). But we have to consider the collection of provenance data during the implementation (or manually annotating the data). Therefore, there is no inherent reliability, but it is possible to reach high reliability using methods known from IT Security explicitly.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a hierarchical framework for provenance based on the fragmentation of provenance information and their reliability. We introduced different layers that complement each other if additional information is available. For instance, the top-most layer shows causal dependencies describing the overall process but we do not premise functional dependency, because of fragmentary knowledge. By contrast, functional dependency is the premise for the second layer. Moreover, we pointed out how existing models and formalisms fit into our framework or address different topics such as causality. Finally, we pointed out limitations of current approaches based on the state of the art.

For future work, we want to consider different aspects of provenance and related topics. For instance, query non-answers [10, 20] are an emerging topic not considered in our framework. From our point of view this is related to causality [25], because non-existing results have no provenance. In this paper, the dominant dimensions for the framework are fragmentation and uncertainty (reliability). However, in future we want to evaluate the interaction of other aspects with our approach and present possible extensions. Additionally, we developed this framework to categorize the most prominent provenance approaches and their limitations regarding the mentioned aspects and encourage the community to address these challenges. Finally, we want to use our framework for designing tailor-made provenance systems based on the requirements of the system.

7. REFERENCES

[1] Acar, U., et al. A graph model of data and workflow provenance. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)* (2010), USENIX, pp. 8/1–8/10.

[2] Agrawal, P., Benjelloun, O., Sarma, A. D., Hayworth, C., Nabar, S., Sugihara, T., and Widom, J. Trio: a system for data, uncertainty, and lineage. In *Demonstration at Proc. Int'l Conf. on Very large data bases (VLDB)* (2006), VLDB Endow., pp. 1151–1154.

[3] Altintas, I., Barney, O., and Jaeger-Frank, E. Provenance collection support in the kepler scientific workflow system. In *Provenance and Annotation of Data*, vol. 4145 of *LNCS*. Springer, 2006, pp. 118–132.

[4] Amsterdamer, Y., Davidson, S. B., Deutch, D., Milo, T., Stoyanovich, J., and Tannen, V. Putting lipstick on pig: Enabling database-style workflow provenance. *Proc. VLDB Endow. 5*, 4 (2011), 346–357.

[5] Amsterdamer, Y., Deutch, D., and Tannen, V. Provenance for aggregate queries. In *Proc. Symposium on Principles of Database Systems (PODS)* (2011), ACM, pp. 153–164.

[6] Biton, O., Cohen-Boulakia, S., and Davidson, S. B. Querying and managing provenance through user views in scientific workflows. Tech. Rep. MS-CIS-07-13, University of Pennsylvania, 2007.

[7] Biton, O., Cohen-Boulakia, S., Davidson, S. B., and Hara, C. S. Querying and managing provenance through user views in scientific workflows. In *Proc. Int'l Conf. on Data Engineering (ICDE)* (2008), IEEE, pp. 1072–1081.

[8] Braun, U., Shinnar, A., and Seltzer, M. Securing provenance. In *Proc. Workshop on Hot Topics in Security* (2008), USENIX, pp. 4:1–4:5.

[9] Buneman, P., Khanna, S., and Tan, W.-C. Why and where: A characterization of data provenance. In *Proc. Int'l Conf. on Database Theory (ICDT)* (2001), vol. 1973 of *LNCS*, Springer, pp. 316–330.

[10] Chapman, A., and Jagadish, H. V. Why not? In *Proc. Int'l Conf. on Management of Data (SIGMOD)*

(2009), ACM, pp. 523–534.

[11] Cheney, J., Chiticariu, L., and Tan, W. C. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases 1*, 4 (2009), 379–474.

[12] Cheney, J., Chong, S., Foster, N., Seltzer, M., and Vansummeren, S. Provenance: A future history. In *Proc. Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2009), ACM, pp. 957–964.

[13] Chiticariu, L., and Tan, W.-C. Debugging schema mappings with routes. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)* (2006), VLDB Endow., pp. 79–90.

[14] Cui, Y., and Widom, J. Lineage tracing in a data warehousing system. In *Proc. Int'l Conf. on Data Engineering (ICDE)* (2000), IEEE, pp. 683–684.

[15] Davies, J., German, D. M., Godfrey, M. W., and Hindle, A. Software bertillonage: Finding the provenance of an entity. In *Proc. Conf. on Mining Software Repositories (MSR)* (2011), ACM, pp. 183–192.

[16] Green, T. J., et al. Orchestra: Facilitating collaborative data sharing. In *Demonstration at Proc. Int'l Conf. on Management of Data (SIGMOD)* (2007), ACM, pp. 1131–1133.

[17] Green, T. J., Ives, Z. G., and Tannen, V. Reconcilable differences. In *Proc. Int'l Conf. on Database Theory (ICDT)* (2009), ACM, pp. 212–224.

[18] Green, T. J., Karvounarakis, G., and Tannen, V. Provenance semirings. In *Proc. Symposium on Principles of Database Systems (PODS)* (2007), ACM, pp. 31–40.

[19] Halpern, J. Y., and Pearl, J. Causes and explanations: A structural-model approach. part i: Causes. *Brit. J. Phil. Sci. 56* (2005), 843–887.

[20] Huang, J., Chen, T., Doan, A., and Naughton, J. F. On the provenance of non-answers to queries over extracted data. *Proc. VLDB Endow. 1* (2008), 736–747.

[21] Ikeda, R., and Widom, J. Panda: A system for provenance and data. *IEEE Data Eng. Bull. 33*, 3 (2010), 42–49.

[22] Lu, R., Lin, X., Liang, X., and Shen, X. Secure provenance: The essential of bread and butter of data forensics in cloud computing. In *Proc. Symposium on Information, Computer and Communications Security (ASIACCS)* (2010), ACM, pp. 282–292.

[23] Lyle, J., and Martin, A. Trusted computing and provenance: Better together. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)* (2010), USENIX, pp. 1/1–1/10.

[24] McDaniel, P., Butler, K., McLaughlin, S., Sion, R., Zadok, E., and Winslett, M. Towards a secure and efficient system for end-to-end provenance. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)* (2010), USENIX, pp. 2/1–2/5.

[25] Meliou, A., Gatterbauer, W., Halpern, J., Koch, C., Moore, K., and Suciu, D. Causality in databases. *IEEE Data Eng. Bull 33*, 3 (2010), 59–67.

[26] Meliou, A., Gatterbauer, W., Moore, K. F., and Suciu, D. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endow. 4*, 1 (2010), 34–45.

[27] Moreau, L. and Freire, J. and Futrelle, J. and McGrath, R. and Myers, J. and Paulson, P. *Open Provenance Model*, 1997.

[28] Mouallem, P., Barreto, R., Klasky, S., Podhorszki, N., and Vouk, M. Tracking files in the kepler provenance framework. In *Scientific and Statistical Database Management*, vol. 5566 of *LNCS*. Springer, 2009, pp. 273–282.

[29] Schäler, M., Schulze, S., Merkel, R., Saake, G., and Dittmann, J. Reliable provenance information for multimedia data using invertible fragile watermarks. In *Proc. Brit. Nat. Conf. on Databases (BNCOD)* (2011), vol. 7051 of *LNCS*, Springer, pp. 3–17.

[30] Simmhan, Y. L., Plale, B., and Gannon, D. Karma2: Provenance management for data-driven workflows. *Int'l J. Web Service Res. 5*, 2 (2008), 1–22.

[31] Tan, V., Groth, P., Miles, S., Jiang, S., Munroe, S., Tsasakou, S., and Moreau, L. Security issues in a soa-based provenance system. In *Provenance and Annotation of Data*, vol. 4145 of *LNCS*. Springer, 2006, pp. 203–211.