Transformation Lifecycle Management with Nautilus

Melanie Herschel and Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen, 72076 Tübingen, Germany
[melanie.herschel | torsten.grust]@uni-tuebingen.de

1. THE TRANSFORMATION LIFECYCLE

When developing data transformations—a task omnipresent in applications like data integration, data migration, data cleaning, or scientific data processing—developers quickly face the need to verify the semantic correctness of the transformation. Declarative specifications of data transformations, *e.g.* SQL or ETL tools, increase developer productivity but usually provide limited or no means for inspection or debugging. In this situation, developers today have no choice but to manually analyze the transformation and, in case of an error, to (repeatedly) fix and test the transformation.

As a simple example, consider a developer who wonders why some products are missing from a SQL query result (making it obvious for him that the query is faulty). Possible reasons abound, *e.g.* were product tuples filtered by a particular selection or are expected join partners missing? Usually, the developer will test several modified versions of the original query, all targeted towards identifying the reason for the missing tuples, for example by relaxing or removing a selection predicate and then observing whether the products appear in the result. The original query is fixed based on the result of this analysis. For instance, the developer may decide to employ a left-outer join instead of a natural join. He then tests the query again (1) to see if his expectations are finally met and (2) to verify that his changes do not introduce unwanted side-effects, *e.g.* return too many products. Developers will typically undergo several such manual *analyze-fix-test (AFT)* cycles before reaching the expected result—a truly tedious, time consuming, and error-prone task.

The need for transformation development support has already been recognized, for instance in data cleaning [17], data integration [1, 5, 20], data extraction [12], and scientific workflows [7]. However, we are not aware of tools that support the *complete* AFT cycle. With respect to data cleaning, ensuring semantic correctness of data transformations helps avoid data errors in the result. It thus complements methods to correct data errors, once they entered a database.

The problem of analyzing, fixing, and testing data transformations not only arises when a transformation is initially developed. Over time, requirements on a data transformation change, *e.g.* new attributes have to be returned due to new laws, autonomous sources change, or customers now require information formatted differently. This evolution of a transformation entails change which, again, requires at least one AFT cycle.

Overall, we observe that from the creation of a transformation to its ultimate retirement when applications no longer rely on it, the transformation undergoes many AFT cycles. We qualify this as the *transformation lifecycle* that, as mentioned previously, is mostly dealt with manually today. No systematic, tool-supported management of the AFT cycle, let alone of a sequence of AFT cycles, possibly spread over a long period of time, exists today.

We advocate that the time is ripe for database researchers to step in and support *transformation lifecycle management (TLM)*. Our vision is that developers leverage semi-automatic tools for TLM to ease the process and reduce the time needed for AFT cycles performed during transformation development or evolution. This paper introduces **Nautilus**[1], our approach to TLM for relational data transformations expressed in SQL. More specifically, after covering related work (Section 2) we discuss (1) the **interactions and the workflow** (Section 3) of the Nautilus system, (2) its **architecture** (Section 4) with (preliminary) **algorithms** employed in the different system components; and (3) a **benchmark** proposal for systems like Nautilus (Section 5). Nautilus is an ongoing and long-term project and the discussion highlights our current status as well as future research challenges and possible solutions to these.

2. RELATED WORK

Throughout the paper, we will include the closest related work directly in the relevant discussion. In this section, we briefly mention further related work pertinent to the three phases of the AFT cycle.

As mentioned above, although the need for transformation development support has been recognized, no tools that support the complete AFT cycle exist. Indeed, previous approaches solely focus on analysis based on sample output data, *e.g.* by leveraging their data provenance [4]. Potter's Wheel [17] additionally supports testing by automatic discrepancy detection. Other approaches analyze the behavior of a data transformation through the generation of test data that exercises all transformation paths [10, 14] or observe the evaluation of transformation subexpressions at arbitrary granularity [9]. Integrated support for the identification of sensible modifications, *i.e.* the fix phase, or the evolution of a transformation over time, is missing, though.

In the context of the fix phase of an AFT cycle, algorithms for schema evolution [6, 16, 21], which adapt a transformation to changes in source schemas, provide support for one specific type of requirement change (*i.e.* a source schema change). Changing a

[1] www.nautilus-system.org

transformation such that it generates additional specific output tuples is addressed in [18] in terms of SQL rewriting.

Concerning the analysis and the test phase, the well-known problems of view update translation [8] and view maintenance [2] are relevant. However, as we shall see in Section 3, Nautilus not only copes with updates to the data, but also with updates to the query.

Having put Nautilus in perspective to existing work, let us focus on the system itself, starting with a description of its workflow.

3. NAUTILUS WORKFLOW

The Nautilus system is intended to support TLM for the particular case of data transformations expressed as SQL queries. Currently, we are focusing on possibly nested SQL queries involving selection, projection, joins (θ-join and outer joins), union, set difference, aggregation and grouping.

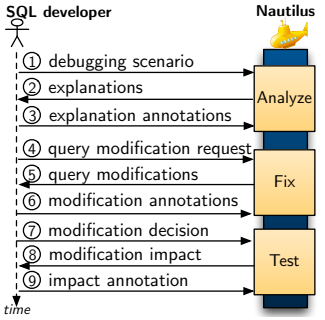

Figure 1: Interactions along one AFT cycle.

The Nautilus workflow throughout one AFT cycle, depicted in Figure 1, consists of an analysis, fix, and test phase. Developers may iterate over the Nautilus workflow just like they do for the manual workflow.

Analyze. First, a developer specifies a *debugging scenario* S (Step ①) that includes (i) SQL queries to be analyzed, (ii) source data, (iii) a description of the desired result, and (iv) further constraints that potentially restrict the set of possible explanations (extends formal definition of [11]). Consider the SQL queries Q_1 and Q_2 over the source instance D that consists of tables R and T (Figure 2). Assume that the developer specifies that both queries should return the same number of tuples (a realistic requirement in a scenario where for every department in the result of query Q_1 there needs to be an associated manager in the result of Q_2, for example). Should that constraint be violated, the developer suspects the error to be in Q_1. He thus declares the result of Q_2 over D, denoted as $Q_2(D)$, as "immutable", meaning that the extension of $Q_2(D)$ should not change during the AFT process. When using Nautilus, a developer simply has to specify what he wants during analysis and returned explanations provide guidance for the fix and test phases. Opposed to that, the old manual process requires him to identify "suspicious" spots of the transformation, to explicitly spell out changes, and to verify that the manual changes do not violate the constraints in debugging scenario S.

R

A	B	C
1	2	3
4	5	6
7	8	6

T

C	D
1	3
2	6
6	8

Q_1:
```
SELECT A, D
FROM R, T
WHERE R.C = T.C AND R.C < 10
```

Q_2:
```
SELECT C
FROM R
WHERE C <10
```

Figure 2: Sample data and queries (note: $|Q_1| = 2 \neq 3 = |Q_2|$).

Explanations returned by Nautilus in Step ② provide information on why the expected result occurred or what may cause a discrepancy between the actual and the expected result. In the former case, data provenance techniques [4] are a natural choice to generate explanations and Nautilus reuses existing techniques. In the latter case, techniques that explain why data is missing from a query result have been proposed. The methods proposed to explain such missing answers either return instance-based [11, 12], query-based [3], or modification-based [18] explanations. Continuing with our example of Figure 2, we observe that the requirement that both queries return the same number of tuples is not met. A possible instance-based explanation points out modifications to the *source data*, e.g., that R needs to contain one more tuple that joins with a tuple in T in order to return three tuples as well. A query-based explanation identifies the *query operator* responsible for filtering tuples from the result, which, in this example is the join between R and T. Finally, a possible modification-based explanation suggests a *query rewrite* such as replacing the join by a left-outer join.

The unification of data provenance of existing data and missing answers is an ongoing effort, for which a model using causality has recently been proposed [13]. Nautilus can benefit from such a unification, but efficient algorithms generating such unified provenance information are yet to be developed.

Explanations cover many different possibilities of why the desired result was (not) computed. Should, based on this information, the developer deem the query to be correct, he is done. Otherwise, if the returned explanations suggest that one of the queries is faulty, a subsequent fix phase becomes necessary. To guide the fix algorithms, the developer attaches a Boolean *annotation to some explanations*, identifying them as being (ir)relevant (Step ③). For instance, he may deem the query-based explanation described above as relevant, but may disagree with the modification-based explanation. This translates his belief that the join operator is indeed the faulty operator, but that a left-outer join is not the solution to the problem. In general, all annotations provided throughout an AFT cycle (*i.e.* explanation annotations and annotations made in subsequent steps) are logged to learn from these for future steps in the same or future AFT iterations.

Fix. In Step ④, the developer issues the *request to generate modifications to the query*. Nautilus then *computes query modifications* (Step ⑤) based on annotated explanations and constraints specified in S. A query modification consists of a rewritten SQL query annotated with changes to the actual query code. Two possible query modifications of Q_1 in our example appear in Figure 3, where changes to the original SQL query (Q_1 in Figure 2) are underlined. These Nautilus-generated query modifcations allow developers to restrict the set of (correct) edits to consider—otherwise, these edits would have to be verified manually.

```
SELECT A, D
FROM R, T
WHERE R.C = D AND R.C < 10
```

(a) Query Modification 1.

```
SELECT A, D
FROM R LEFT OUTER JOIN T ON R.C = T.C
WHERE R.C < 10
```

(b) Query Modification 2.

Figure 3: Query modifications generated for Q_1 (edits marked).

Similarly to explanation annotation, the developer has the possibility to *annotate query modifications* (Step ⑥). These annotations are considered in subsequent iterations of the AFT cycle, for instance, to better rank returned query modifications or to avoid recomputing irrelevant modifications again. In our example of Figure 3, let Query Modification 1 be annotated as relevant, whereas Query Modification 2 is irrelevant.

Test. Based on his analysis and annotation of modifications, the developer now has gained sufficient knowledge to modify the actual query according to his requirements. In our simple example, the *modification decision* (Step ⑦) simply corresponds to deciding that Query Modification 1 yields the new query. However, in a more complex setting, such a direct correspondence between a candidate

query modification and the actual revised query does not necessarily exist. For instance, if the developer annotates both modifications of Figure 3 as relevant, the final query he derives may be the query Q_1' depicted below, in which both the join type as well as the join predicate have changed compared to the original query Q_1.

```
Q1': SELECT A, D
     FROM R LEFT OUTER JOIN T ON R.C = D
     WHERE R.C < 10
```

The new query is now tested w.r.t. the constraints defined in debugging scenario S. If constraints are violated, Nautilus returns corresponding error messages. Moreover, Nautilus determines the modification's *impact* on the query result. Indeed, although all constraints are satisfied, *e.g.* the number of tuples returned by Q_1' and Q_2 is the same, the values in $Q_1'(D)$ now differ from previous values. Nautilus indicates this impact by reporting statistics such as "one tuple added to $Q_1(D)$", "100% of new values in column `D`", "new values in column `D` range from `3` to `6`" or "number of distinct values in column `D` equals two". It is important to be aware of such effects, *e.g.* when the result of Q_1 is the input to another query. Therefore, Nautilus determines the difference between an original query result $Q(D)$ and its modified version $Q'(D)$ (Step ⑧).

Such statistics provide a developer with valuable information to verify that query changes have no adverse impact on query results. Without Nautilus, a developer would have to study this impact manually and, on large data sets in particular, it is easy to overlook differences in comparison to previous results.

The final step is *impact annotation* (Step ⑨), where a developer marks data changes as acceptable or unacceptable. The developer will not be notified of acceptable changes in subsequent iterations of the AFT cycle. Unacceptable query result changes, however, translate to new constraints that are added to scenario S. In our example, a developer may determine that a minimum value of 3 in column `D` is unexpected thus flags this impact as unacceptable.

4. ARCHITECTURE

Figure 4 shows the Nautilus architecture, which embeds components for the three phases of the AFT process. The *explanation manager* takes care of all steps relevant for analysis whereas the *query modification manager* is responsible for generating, annotating, analyzing, and ranking query modifications during the fixing phase. The *development cycle manager* serves as an interface between the *explanation manager*, the *query modification manager*, and the *metadata repository*. It thus has a global view of the current state of the development process, which is necessary during the test phase. Therefore, the *development cycle manager* additionally encompasses all components relevant to the test phase.

Nautilus accesses source data stored in a relational database (*DB*). This database may be (a sample of) a production database,

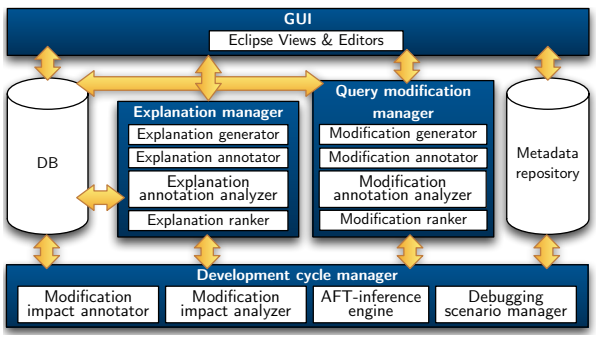

Figure 4: Nautilus Architecture

providing example data to analyze the behavior of the transformation. As such example data may not cover all cases and some errors may go undetected, algorithms to generate data to analyze the query semantics and that cover all test cases have been proposed [10, 14]. Such techniques are complementary to Nautilus and are not in the focus of this paper.

The *metadata repository* persists all data and metadata that Nautilus saves through one or more AFT iterations. This information is accessed by the *development cycle manager* and passed on to the *explanation manager* and the *query modification manager* that require these during their respective computations. As the information stored highly depends on the algorithms used, we do not provide a detailed discussion here. Essentially, annotations and a history of decisions and performed computations are stored in the *metadata repository*. Interesting issues are the efficient and effective storage of meta-data, and its efficient retrieval.

The *graphical user interface (GUI)* lies between the Nautilus user and the algorithms underlying Nautilus. Our Nautilus prototype is implemented as an Eclipse plugin. For sceenshots, please visit the project website.

We now discuss selected Nautilus components in more detail, beginning with the explanation manager. We leave out those components whose purpose is obvious and that do not require solving new challenges (i.e., ranker components, the GUI, and the debugging scenario manager). Note that the architecture is intended to be open and to provide interfaces to easily plug in individual algorithms and solutions.

4.1 Explanation Manager

Explanation generator. As discussed in Section 3, Nautilus returns explanations w.r.t. a debugging scenario S as provenance information of existing or missing data. Let us discuss two issues and our current solutions that arise for explanation generation, i.e., *debugging scenario expressiveness* and *explanation scope*.

Considering the expressiveness of debugging scenarios, methods for data provenance of existing data allow to compute the provenance of (sets of) tuples resulting from relational queries [4]. Our system prototype reuses the lineage tracing capabilities of Trio [19] and returns, given a set of tuples $T \in Q(D)$, the actual source tuples that contribute to tuples in T. However, what about questions of the form "Why are there more than 10,000 tuples where attribute `A` is null?", or "Why is the average temperature in California less than the average temperature in Germany?". Such analytical provenance queries are not directly supported by todays provenance systems but are of great interest especially when dealing with large query results where users cannot manually analyze the provenance of every single relevant tuple. Answering such analytical questions requires the computation of the provenance of tuples, the aggregation of provenance information, and the comparison of (aggregated) provenance information.

Current approaches to compute the provenance of missing data [3, 11, 12, 18] explain why one or more tuples, the so-called *missing-answers*, do not exist in query results. Currently, Nautilus supports the Artemis algorithm we devised [11], Missing-Answers [12] and Why-Not [3] and we plan to include ConQueR [18] in the near future. As all these algorithms do not yet support all types of queries in the scope of Nautilus (especially non-monotonic queries and nested queries are not yet supported) and efficiency improvements are necessary to make any of these approaches usable in practice, we are actively working on algorithms to fill these gaps when explaining missing-answers. However, missing-answers are not the only conceivable non-observed

fact. For instance, developers may question the correctness of a combination of attribute values in a tuple (e.g., an employee has apparently been joined to the wrong department) or wonder why, given that employee A is in the result, why his colleague B is not. Besides methods to explain these complex observations, it is also important to study how to enable users to specify such questions. Clearly, a large enough set of questions should be expressible, but the way to express them should not be too complex, so as to require a debugger of its own.

Let us now focus on explanation scope. Whereas algorithms to compute the provenance of existing data consider D, Q, and $Q(D)$ to be fixed, algorithms determining the provenance of missing data make varying assumptions about which of these components may require changes. Whereas Why-Not considers D and Q as fixed to generate query-based explanations, Artemis and Missing-Answers assume only Q to be fixed and compute instance-based explanations. Finally, ConQueR only fixes D and produces modification-based explanations that modify Q. Hence, it is possible that one type of explanation can be computed while another type cannot. For instance, assume that on the output of Q_2 in Figure 2, a developer wonders why none of the tuples that Q_2 returns has a value for $\texttt{C} = 1$. Whereas an instance-based explanation indicating that a tuple containing $\texttt{C} = 1$ is missing from $\texttt{R}$ can be computed, no query- or modification-based explanation is possible.

We propose a new type of explanation, named *hybrid explanation*, that prevents a user from worrying about the above differences in the scope of each individual algorithm. Thus, no matter the debugging scenario, Nautilus returns the most complete set of explanations, regardless of their type. This complete coverage goes beyond unifying all explanations of different types. In fact, new explanations are possible when allowing, for example, a combination of an instance-based and a modification-based explanation. For instance, to explain why tuple (11) is missing from $Q_2(D)$ and assuming Q_2 is no longer fixed, we need a modification to table $\texttt{R}$ that inserts a tuple with $\texttt{C} = 11$. But additionally, we have to change the selection predicate $\texttt{C} < 10$ to a predicate that lets the desired tuple pass. To compute hybrid explanations, we are currently developing the Conseil algorithm[2]. Given a SQL query Q (that may include nested queries and set difference as opposed to previous algorithms explaining missing-answers), a missing-answer t, $Q(D)$ and D, Conseil computes the set of all possible explanations, i.e., the union of all instance-based, query-based, and hybrid explanations. While details of the algorithm are out of the scope of this paper, for the purpose of our subsequent discussion of the fix phase, we nevertheless mention that Conseil computes these explanations by transforming an annotated logical query tree representation determined based on Q and t into an analogous tree of a tuple similar to t but existing in $Q(D)$. Intuitively, Conseil thus "tracks" similar tuples in $Q(D)$ and analyzes why t does not follow the same path. The query tree transformation is determined using a tree-edit-distance-like algorithm and the edit script translates into an explanation.

Explanation annotator. This component links annotations provided by a developer in Step ③ of Figure 1 to explanations computed by the explanation generator. Our current implementation considers a Boolean annotation that allows developers to mark explanations as relevant or irrelevant w.r.t. the debugging scenario. Marking an explanation as relevant translates a developer's belief that the explanation exposes an interesting fact that warrants closer analysis. On the contrary, all facts exposed by irrelevant explanations should not be further explored. These annotations are used as guidance on how the space of all possible explanations is explored and exposed to the developer. To this end, we translate them into probabilities of their relevance (e.g., 1 and 0 for relevant and irrelevant, respectively).

Not all explanations need to be annotated by a developer. For those not explicitly annotated, an annotation of unknown is assumed or, as discussed next, a derived annotation is determined. Explanations with an unknown annotation have a probability of relevance between those of relevant and irrelevant annotations.

Annotations managed by the explanation annotator serve as input to various Nautilus components: the explanation annotation analyzer uses these to derive further annotations. The explanation ranker requires annotations to compute a better rank for explanations. Finally, the explanation generator may use annotations to either prune the computation of explanations (i.e., those explanations that are derived to be irrelevant) or to prioritize the computation of some explanations (based on derived relevant annotations). Finally, the development cycle manager disseminates annotations to the metadata repository and the query modification manager.

Explanation annotation analyzer. Based on annotations collected by the explanation annotator, the explanation annotation analyzer derives information concerning "related" explanations. To illustrate, given that the query-based explanation that identified the join in Q_1 (Figure 2) as faulty has been marked as irrelevant by the user, the explanation annotation analyzer decides that all modification-based explanations that modify the join operator are irrelevant as well (Example 1). On the other hand, if the developer annotated as relevant an instance-based explanation that describes that tuples from $\texttt{R}$ that join with tuples in $\texttt{T}$ are missing, then the explanation annotation analyzer derives that most likely, the query-based explanation that returns the join operator is relevant, too (Example 2). We distinguish the following types of derived annotations.

- **Boolean-valued derived annotations** correspond to relevant or irrelevant explanations. These are derived when, given one annotation, the same (or opposite) annotation needs to be valid for related explanations. Example 1 above illustrates a Boolean-valued derived irrelevant annotation.
- **Real-valued derived annotations** correspond to a probability indicating how relevant an annotation is, given a set of related annotations. Example 2 above illustrates a real-valued derived relevant annotation.

Currently, we model explanations and relationships between them as a Bayesian network [15]. The nodes in the network represent (sets of) explanations. The probability of each node translates the relevance of the corresponding explanations. We are currently studying different conditional probability distribution functions. Further work will focus on incrementally and efficiently adapting the network based on new user input and as new explanations are added, occurring for instance when computing explanations incrementally for efficiency and interactivity reasons.

Derived annotations are fed back to the explanation annotator. One use of derived annotations is determining the priority of explanations, which is part of explanation generation. Essentially, the higher the probability that a non-computed explanation is relevant based on derived annotations, the sooner it should be computed.

4.2 Query Modification Manager

The query modification manager takes as input a debugging scenario, explanations, and their annotations (passed to the development cycle manager by the explanation annotator). When available, it further considers modification annotations that the modification annotator manages. The modification annotator, the modification

[2] Conseil was a passenger of the Nautilus in Verne's novel and means *advice* in French.

annotation analyzer, and the modification ranker are analogous to their counterparts in the explanation manager, so we concentrate our discussion on the modification generator.

The modification generator produces one ore more query modifications that consist of an SQL rewrite, as we already illustrated in Figure 3.

ConQueR [18] is an algorithm that returns modification-based explanations in form of rewritten SQL queries. Obviously, these can trivially be transformed into query modifications defined above. ConQueR applies when tuples are missing from a monotonic query, a limitation we plan to overcome by exploiting the output produced by Conseil. Furthermore, the set of possible query transformations is limited to changing the set of selection predicates, the set of join predicates, or the set of joined tables. That is, ConQueR can produce the query modification shown in Figure 3(a), but it cannot determine the query modification depicted in Figure 3(b). Also, ConQueR assumes the source data to be fixed, so if the data necessary to produce the missing tuple t does not exist, no modification-based explanation or, equivalently, no query modification is returned.

Based on Conseil's computation of hybrid explanations, we are investigating how to determine corresponding query modifications, again using the idea of edit scripts between logical query trees. Unlike for explanation generation, where we are given a source tree t and a target tree of a tuple similar to t (cf. Section 4.2), we are facing the problem that there is no available target tree when computing query modifications (as the target tree will correspond to the query modification). We thus have to explore and prune the search space of all possible target trees and the result should correspond to the target tree that has a minimal distance to the source tree. In summary, we solve a search problem to determine the cheapest tree transformation, given a set of possible tree edit operations and a cost model. Clearly, efficiency is a major concern.

4.3 Development Cycle Manager

The development cycle manager serves two purposes: (i) it manages the test phase of the AFT cycle using the modification impact analyzer and the modification impact annotator, and (ii) it maintains information necessary across the explanation manager and the query modification manager. More specifically, it processes annotations and the current debugging state using the AFT-inference engine to determine possible implications of these to the whole debugging process. It also maintains the debugging scenario itself. We briefly describe details of each component in the remainder of this section, except for the modification impact annotator, as it is analogous to the explanation annotator discussed in Section 4.1. Note that the development cycle manager does not incorporate an impact ranker. The reason for this is that the descriptions of changes in the result data are classified into different categories (error, warning, irrelevant) by the modification impact analyzer and do not require sophisticated and adaptive ranking functions.

Modification impact analyzer. This component tracks and reports the impact of a query modification, once a developer has reached a modification decision. More specifically, given an original query Q that has been modified to query Q', the impact consists of all differences between $Q(D)$ and $Q'(D)$. Essentially, it comprises the side-effects that affect the changed query's result. Note that this kind of side-effect is different from side-effects from view update [8] as, in the case we consider, a side-effect is caused by changes to the query, not by changes to the data.

Both ConQueR [18] and Artemis [11] consider side-effects during explanation generation and aim at minimizing these for monotonic queries, i.e., they minimize $|Q'(D) \setminus Q(D)|$. The modification impact analyzer does not pursue the minimization goal but reports changes between query results based on heuristics. These heuristics build on various statistics (e.g., the number of new tuples, percentage of new values, number of distinct values, as illustrated in Section 3) that the modification impact analyzer collects, updates (when possible, incrementally), and interprets. More specifically, it first identifies which tables, attributes, or sets of tuples are potentially affected and how they may be affected. For instance, when performing query modification 2 on Q_1, we know that all additional values for column `D` are null values. This knowledge allows us to limit the computations in the second step, where we update relevant statistics only, e.g., the frequency of null values in column `D` have changed, whereas the average value of `D` is not affected. Finally, the updated statistics are interpreted by the modification impact analyzer, which decides what changes are significant. For instance, a change in attribute `A` may be translated into x new values in `A` ranging from y to z, whereas the presence of new null values in `D` is not signaled to the user. The distinction between relevant and irrelevant impact is determined by heuristics that can be configured by a user. Essentially, these heuristics allow to classify updated statistics as impact that should be reviewed by the developer (error), notifications that may be interesting (warning), and irrelevant updates.

AFT inference engine. The AFT inference engine gathers annotations and considers the processing state of each component to infer implications of these on the subsequent debugging process. It then dispatches these inferred actions across the different components. Currently, we envision a rule-based inference engine. To better appreciate the role of the AFT inference engine, consider the three examples below.

- **Adaptive debugging scenario.** The AFT inference engine adapts the debugging scenario S based on annotations. For instance, impact annotations that mark some changes to the query result as unacceptable are processed by the AFT inference engine that decides that forbidding these should be part of S.
- **Adaptive ranking.** Based on explanation annotations that designate explanations as relevant, the AFT inference engine determines what query modifications become more likely, based on these explanations. This translates to real-valued relevant annotations for query modifications.
- **Computation pruning.** When a user annotates query modifications as irrelevant, the inference engine determines which explanations will produce equivalent modifications, when possible. Clearly, these explanations need not be computed. This information is propagated to the explanation annotation analyzer in form of new Boolean-valued irrelevant annotations.

Derived information is used both to improve the overall efficiency of the debugging process as well as the user experience. Opposed to the respective annotation analyzers and rankers of the explanation manager and the query modification manager, the actions inferred by the AFT inference engine are not local to a component, but focus on cross-component relationships.

5. BENCHMARK

In the previous sections, we discussed how to use Nautilus and how the system is built. To study its usability for practical debugging scenarios, we propose to benchmark Nautilus (and further systems with similar capabilities) along three dimensions, namely the efficiency, the effectiveness, and the user friendliness on a standardized set of debugging scenarios using predefined tasks and evaluation metrics.

What the benchmark provides. First, debugging scenarios need to be defined. TPC-H data and queries may serve as source data

Task	Description
Task 1 Input support	Given a set of debugging scenarios $\mathcal{S} = \{S_1, \ldots, S_n\}$, compute a set of explanations/query modifications O_i for the i-the debugging scenario. For unsupported debugging scenarios, let $O_i = null$.
Task 2 Output computation	Given a single debugging scenario, compute the complete set of explanations/modifications possible (i.e., no pruning). Sort these according to the ranking function.
Task 3 Impact classification	Given a single debugging scenario, compute the impact of the query modification that corresponds to the final query (as defined by the gold-standard) and classify changes as errors, warnings, and irrelevant changes.
Task 4 User interaction	Given a debugging scenario of the benchmark and the output of an algorithm (explanation, query modification, or impact), let a set of expert users navigate through and annotate produced results until the correct result, as specified by the benchmark is identified.
Task 5 AFT processing	Given a debugging scenario of the benchmark, let a set of expert users iteratively perform the analyze, fix, and test phases until the correct result as specified by the benchmark is obtained.
Task 6 Installation	Let non-expert users install and configure the system so that all applicable debugging scenarios of the benchmark can be applied. Let the user fill out the standardized installation form provided by the benchmark.
Task 7 System mastering	Given a working installation of the system, let non-expert users learn how to use the system. A user reaches the level of an expert user when he proves to be able to process at least three randomly selected debugging scenarios of the benchmark that can be applied to the system. Let the new expert user fill out the standardized initiation form provided by the benchmark.
Task 8 User satisfaction	Let expert users fill out the standardized user satisfaction form provided by the benchmark.

Table 1: Benchmark tasks

and correct queries. The queries are then modified to obtain a faulty version (this procedure has already been used in [11, 18]). In designing debugging scenarios for the benchmark, we need to take special care in covering all interesting cases. In addition to generated data, the benchmark should provide real-world data and queries. Obtaining both is not trivial but first steps in this direction have already been undertaken [18].

Additionally, the benchmark needs to provide a gold-standard for the results of different algorithms, given the debugging scenarios. When proceeding as described above for the TPC-H data, the correct query is known, and the gold-standard for explanation generation, query modification, and impact can be derived accordingly. When no queries are available, which is often the case for real-word data, the correct query needs to be defined in addition to the query being debugged, before proceeding with the creation of explanations, modification, and impact results.

Finally, to study user satisfaction and usability, we envision the creation of dedicated surveys to be added to the benchmark.

Benchmark tasks. Table 1 summarizes the tasks we define in our benchmark. We distinguish three dimensions for the evaluation of systems like Nautilus: effectiveness (Tasks 1–3), efficiency (Tasks 2–5), and user friendliness (Tasks 6–8). Whereas the first two dimensions can be measured by adequate metrics, the third dimension is more subjective and requires user studies.

Evaluation metrics. Given the output of Task 1 we compute the coverage of supported debugging scenarios over a given set of debugging scenarios $\mathcal{D}$ as $coverage(\mathcal{D}) = \frac{|\{O_i | O_i \neq null\}|}{|\mathcal{D}|}$. Given the output of Task 2, we compute the recall, precision, and f-measure over the sets of ranked explanations/query modifications. To evaluate the output of Task 3, we compute the same measures for each individual class as well as over all classes. The latter gives us a measure of what changes have been successfully detected during impact analysis and abstracts from the heuristics used for classification.

To measure efficiency, we propose the three metrics time, click count, and annotation count. Note that these are also employed when evaluating user friendliness. In this case, the measurements complement the survey-based results.

6. CONCLUSION

This paper introduced Nautilus, a system to semi-automatically support developers in designing, understanding, refining, and debugging data transformations, in particular SQL queries. Nautilus supports the three phases of the currently manual development process, i.e., the analysis, fix, and test phases through adequate algorithms and tools. Using Nautilus, the data quality of data entering a database through a data transformation (e.g., an ETL process to obtain data to be stored in a data warehouse) can potentially be significantly improved by recognizing errors in the transformations that yield data errors. Thus, transformation lifecycle management is a technique allowing error avoidance, as opposed to techniques for error correction such as entity resolution.

We described the general workflow when using Nautilus and discussed its architecture. For various components of the architecture, we presented (preliminary) solutions to implement the desired functionality. We also presented a possible benchmark to evaluate algorithms relevant to different steps of the AFT cycle.

The research on Nautilus is ongoing work, and in the future, we plan to live up to the vision of Nautilus given in this paper. To do so, next steps include extending algorithms for the analysis phase, refining existing algorithms and devising new algorithms for the remaining phases, and a proper comparative evaluation using our benchmark.

7. REFERENCES

[1] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: Mapping understanding and design by example. In *ICDE*, pages 10–19, 2008.

[2] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, 1986.

[3] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD Conference*, 2009.

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

[5] L. Chiticariu and W. C. Tan. Debugging schema mappings with routes. In *VLDB*, pages 79–90, 2006.

[6] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the PRISM workbench. *Proceedings of the PVLDB Endowment (PVLDB)*, 1(1), 2008.

[7] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD Conference*, pages 1345–1350, 2008.

[8] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3), 1982.

[9] T. Grust, F. Kliebhan, J. Rittinger, and T. Schreiber. True Language-Level SQL Debugging. In *EDBT Conference*, 2011.

[10] B. P. Gupta, D. Vira, and S. Sudarshan. X-data: Generating test data for killing SQL mutants. In *ICDE Conference*, 2010.

[11] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *PVLDB*, 3(1), 2010.

[12] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1), 2008.

[13] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 2011.

[14] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD Conference*, 2009.

[15] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of plausible inference*. Morgan Kaufmann Publishers, 2nd edition, 1988.

[16] Y.-G. Ra and E. A. Rundensteiner. A transparent schema-evolution system based on object-oriented view technology. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):600–624, 1997.

[17] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB Conference*, 2001.

[18] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *SIGMOD Conference*, 2010.

[19] J. Widom. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*. Springer, 2008.

[20] L.-L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD Conference*, 2001.

[21] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *International Conference on Very Large Databases (VLDB)*, 2005.