The Nautilus Analyzer: Understanding and Debugging Data Transformations

Melanie Herschel, Hanno Eichelberger

The Nautilus Analyzer: Understanding and Debugging Data Transformations

Melanie Herschel
Université Paris Sud / INRIA Saclay
91405 Orsay Cedex, France
melanie.herschel@lri.fr

Hanno Eichelberger
Universität Tübingen
72076 Tübingen, Germany
hanno.eichelberger@student.uni-tuebingen.de

ABSTRACT

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.

The goal of the Nautilus project is to semi-automatically support this analysis-fix-test cycle. This demonstration focuses on one main component of Nautilus, namely the Nautilus Analyzer that helps developers in understanding and debugging their data transformations. The demonstration will show the capabilities of this component for data transformations specified in SQL on scenarios from different domains that are based on real-world data.

We provide an overview the Nautilus Analyzer, discuss components and implementation techniques, and outline our demonstration plan. The Nautilus website (`http://nautilus-system.org`) features a video, screenshots, and further details.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms

Keywords

data provenance, query analysis

1. NAUTILUS ANALYZER OVERVIEW

The Nautilus system [4] aims at supporting developers during data transformation development by providing semi-automatic tools for the otherwise manual analyze, fix, and test tasks that they usually perform. Due to the lack of space, we refer readers to [4] for details, as in this section, we focus on illustrating the benefits, the workflow, and the components of the Nautilus Analyzer, the subject of the proposed demonstration. As the name suggests, the Nautilus Analyzer manages the analysis phase of the process.

Consider the source tables R and T as well as queries Q_1 and Q_2 in Fig. 1. Assume that the developer specifies that both queries should return the same number of tuples (a realistic requirement for instance in a scenario where every department in the result of Q_1 needs to be associated with a manager in the result of Q_2). Should that constraint be violated, the developer suspects the error to be in Q_1. He thus declares the result of Q_2 over source instance D, denoted as $Q_2(D)$, as "trusted", meaning that the extension of $Q_2(D)$ should not change during the analysis-fix-test process. For analysis, the above specification translates to the questions *"Why are there too many tuples in the result of Q_1?"* (Why-question) or *"Why are tuples missing from the result of Q_1?"* (Why-Not question).[1]

Essentially, when using Nautilus, a developer specifies what he wants during analysis in the form of a *debugging scenario*, as illustrated above. The result of analysis is a set of *explanations* that provide guidance for the fix and test phases. These explanations take the form of provenance information, i.e., the Nautilus Analyzer relies on Why-provenance [2] and Why-Not provenance [1, 5, 7, 9] to generate explanations answering the respective types of questions. Without Nautilus, the traditional manual process requires the developer 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 the debugging scenario.

Continuing our example of Fig. 1, we observe that the requirement that both queries return the same number of tuples is not met. Therefore, the developer asks a Why-Not question on the result of Q_1. The following answers are possible: (i) R needs to contain one more tuple that joins with a tuple in T in order to return three tuples as well (instance-based explanation [5, 7]), (ii) the join between R and T is the operator filtering tuples from the result (query-based explanation [1]), or (iii) the query would produce the desired result if the join was replaced by a left-outer join (modification-based explanation [9]). Let us denote these explanations as X^{IB}, X^{QB}, and X^{MB}, respectively.

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. During the fix

[1]Processing the scenario without trusting any table would result in a larger set of Why and Why-Not questions to be answered.

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

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.
ACM 978-1-4503-1156-4/12/10.

phase, Nautilus suggests changes to the queries in order to meet the requirements specified in the debugging scenario. To guide the fix algorithms, the developer, during analysis, may attach a Boolean *annotation to some explanations*, identifying them as being (ir)relevant. 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. The Nautilus Analyzer exploits these annotations in subsequent steps in the same or future analysis-fix-test iterations.

As the number of explanations can become quite large, Nautilus also provides means on ranking these. Our ranking model takes into account both heuristics and explanation annotations, e.g., if the user marks the instance-based explanation above as interesting, similar and related explanations will be ranked higher.

The Nautilus Analyzer consists of two main components to implement the above functionality: a *graphical user interface (GUI)* to graphically and intuitively specify a debugging scenario and to interact with produced explanations and the *explanation manager*. Due to the lack of space, we focus the technical discussion on the explanation manager and refer interested readers to the project website for screenshots and a video demonstrating the GUI.

2. THE EXPLANATION MANAGER

The input of the explanation manager consists of a debugging scenario and optional developer-provided Boolean annotations. The *explanation graph* is the central data model that represents all explanations. Nautilus initializes and refines the graph based on interactions with the four main components of the explanation manager: (i) the *explanation generator*, (ii) the *explanation annotator*, (iii) the *annotation analyzer*, and (iv) the *explanation ranker*.

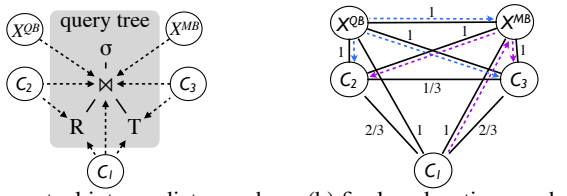
Explanation generator. Given a debugging scenario, the explanation generator returns a set of explanations. In general, a debugging scenario $S = \{\mathbf{Q}, D, \mathbf{Q}(D), E, C\}$ includes a set of SQL queries $\mathbf{Q}$ to be analyzed, the source instance D, the set of query results $\mathbf{Q}(D)$, the set of tuples (existing or missing) to be explained, denoted as E, and a set of constraints C. In our current implementation, C is limited to trusting or minimally affecting tables in D or query results in $\mathbf{Q}(D)$. This definition of a debugging scenario naturally extends its definition in [5]. In our example, the debugging scenario is defined by $\mathbf{Q} = \{Q_1, Q_2\}, D = \{R, T\}, \mathbf{Q}(D) = \{Q_1(D), Q_2(D)\}$, $E = \{\text{missing tuple from } Q_1(D)\}$, and $C = \{\text{trust on } Q_2(D)\}$.

The debugging scenario translates to a (set of) Why or Why-Not provenance questions. Each question is then processed using one or more of the implemented algorithms. These include Why-Not [1], ConQuer [9], Missing-Answers [7], Artemis [5], and lineage [3][2]. The result of each algorithm is generalized as a set of explanations, which form the input to the explanation graph initialization. In our example, we return the Why-Not explanations mentioned in the introduction.

Explanation graph. To initialize the explanation graph, we conceptually proceed as follows. Please note that our implementation does not strictly follow this descriptions for efficiency reasons.

Conceptually, we first consider a logical operator tree representation, the *query tree*, of the queries to be analyzed, e.g., of Q_1. Each computed explanation results in an *explanation node* that connects to nodes in the query tree via edges directed from the explanation

[2]The authors of [1, 9] kindly provided us with their code, which significantly facilitated the re-implementation and integration into our Nautilus Eclipse plugin. As for lineage, we reuse the lineage capabilities of the Trio system [10].



(a) conceptual intermediate graph (b) final explanation graph

Figure 2: Creating the explanation graph

node to query tree nodes. The edge between a query-based explanation (e.g., X^{QB}) simply connects the explanation node to the query tree node that represents the identified culprit operator, whereas a modification-based explanation (e.g., X^{MB}) connects to the query tree nodes affected by the modification. As for instance-based explanations, we create equivalence classes based on what we call "join patterns". To illustrate these, let us assume that no trust lies on $Q_2(D)$ and hence, T is subject to modification. Then, three possible patterns for instance-based explanations exist: (i) insert into R and T s.t. the join is satisfied, (ii) insert to R s.t. the inserted tuple joins with an existing tuple in T, and (iii) insert to T s.t. the inserted tuple joins with an existing tuple in R. This results in three equivalence classes which we denote C_1, C_2, and C_3. Explanations within an equivalence class only differ w.r.t. the existing tuples involved and convey the same information. Thus, we regard these as equivalent. For each equivalence class, we create a *class node* in the graph. The class node connects to each query tree node that either represents a source table to which an insertion applies or that represents an unfulfilled join. Individual instance-based explanations appear in the explanation graph as nodes that simply connect to their respective class node. Considering all three types of Why-Not explanations of our example, at this point of conceptual processing, we obtain the graph shown in Fig. 2(a) (omitting the individual instance-based explanation nodes for conciseness).

In a second step, we abstract from the query nodes and directly connect explanations with weighted edges. The weight corresponds to explanation similarity. To determine the similarity of two class nodes, we compute the Jaccard coefficient of query tree nodes reached from the pair of compared explanations. For all remaining pairwise combinations of explanation types, the Jaccard coefficient is solely based on the inner query nodes (i.e., source tables are no longer considered). Fig. 2(b) shows the resulting explanation graph (individual instance-based explanations, again omitted, are all considered equivalent, so their similarity within an equivalence class equals 1, and their similarity across equivalence classes is equal to the similarity of the respective equivalence classes). Please ignore the dashed edges for now.

Explanation annotator. Annotations are either user defined or input by other Nautilus components[3]. The goal of the explanation annotator is to link annotations provided by a developer through the GUI to explanations in the explanation graph. We support two user-annotations, i.e., "relevant explanation" and "irrelevant explanation". A developer provides these for an explanation if he decides that the inspected explanation provides valuable information in the context of the debugging scenario or not, respectively. Note that not all explanations need to be annotated by a developer. For those not explicitly annotated, we assume an annotation of "unknown" or, as discussed next, derive an annotation using the explanation annotation analyzer.

[3]At the current development stage, the Nautilus Analyzer only supports user-defined annotations and annotations derived by the annotation analyzer. In the future, the fixing and testing components of Nautilus may contribute further annotations.

Internally, the Nautilus Analyzer translates annotations to probabilities that indicate the probability that an explanation pinpoints the problem to be detected by the developer. A user-annotation of "relevant explanation" translates into a probability of 1 whereas "irrelevant explanation" corresponds to a value of 0. Whenever no annotation (user- or system-provided) is available, a default probability (currently 0.5) is assigned to an explanation.

Annotation analyzer. Based on annotations collected by the explanation annotator, the explanation annotation analyzer derives further probabilities for explanations similar to the annotated ones. To this end, it spans a Bayesian network [8] over the explanation graph as follows: user-annotated explanations serve as source nodes that provide the prior probabilities. Starting from these nodes, we then traverse the explanation graph in breadth-first, only following edges whose attenuated similarity is above a given threshold θ. The attenuated similarity of an edge e on a path of length k starting at a user-annotated explanation is calculated as $\Pi_{1\leq i\leq k} sim_i \times \alpha^{k-1}$, where $0 \leq \alpha \leq 1$ denotes an attenuation factor. We compute conditional probabilities for all explanations reached during this traversal procedure. The details on the conditional probability functions are out of the scope of this paper, essentially, we have defined a function for each combination of source and target explanation type. The functions either return a Boolean or a real-value.

Fig. 2(b) shows an example of the network we span assuming X^{QB} (blue dashed edges) and an explanation in C_1 (purple dashed edges) are annotated as irrelevant and relevant, respectively, $\theta = 0.7$ and $\alpha = 0.8$. Intuitively, the fact that the query-based explanation is invalidated also invalidates the modification-based explanation on the same query operator, e.g., we propagate a probability of 0 to X^{MB}. However, C_1 being relevant indicates that the join operator is after all relevant and hence a probability of, e.g., $\frac{1}{3}$ propagates X^{MB} whose probability is then re-evaluated.

Explanation ranker. From a user perspective, it is crucial that he does not drown in an overwhelmingly large number of explanations and that the analysis of few explanations is rewarding, i.e., relevant explanations are soon detected. To this end, Nautilus incorporates an explanation ranker. Initial ranking is based on pre-defined heuristics but as the developer provides annotations, the ranking adapts to these to favor explanations with higher probabilities.

Ranking heuristics for Why-provenance include ordering explanations by (i) number of tuples involved, (ii) similarity, and (iii) diversity. Whereas the first criterion is self-explanatory, let us briefly explain the rationale behind the other two heuristics. Similarity will sort similar explanations next to each other (calculated based on shared tuples and query operators involved) whereas diversity aims at presenting different explanations consecutively. Intuitively, similarity is a good choice for eager annotators whereas diversity represents a better choice for lazy annotators. For Why-Not explanations, we additionally consider ranking based on side-effects an instance-based or modification based explanation may have (i.e., changes in query results if tuple insertions or query modifications were actually performed, which they are of course not during analysis). In our example, an initial ranking of instance-based explanations based on diversity for instance returns as top-3 an explanation from C_2, an explanation from C_3, and an explanation from C_1.

In summary, the explanation ranker module is still in a preliminary state compared to the remaining components, as the current ranking schemes only represent a first step towards a unified ranking of explanations (without the user specifying the function). The research on explanation ranking is an ongoing effort and we plan on incorporating more sophisticated ranking algorithms in the future.

3. DEMONSTRATION PLAN

The Nautilus system, and hence its Analyzer component, is implemented as an Eclipse plugin in Java 1.5. Using the Eclipse platform has two significant advantages: first, it allows us to seamlessly integrate the Nautilus components in a platform with which developers are already familiar with; second, it comes with a rich set of plugins that we can leverage. For instance, the demonstration we propose makes use of and extends the Data Tools Platform (`http://www.eclipse.org/datatools/`) and the Zest Graph Visualization Toolkit (`http://www.eclipse.org/gef/zest/`). Note that the Nautilus Analyzer significantly extends the Eclipse plugin we demonstrated for Artemis [6], an algorithm we developed to generate instance-based explanations for Why-Not provenance [5]. Screenshots and a video are available on the project website.

At the conference, we will demonstrate the Nautilus Analyzer using several debugging scenarios and at least three use cases. The first use-case reuses the sample CRIME scenario provided by the Trio system(`http://infolab.stanford.edu/trio/`). The second scenario comes from the government domain using publicly available data about US Congressmen(`http://bioguide.congress.gov`), Spendings(`http://usaspending.gov`), Earmarks (`http://earmarks.omb.gov`), etc.. The third scenario relies on IMDB movie data (`http://www.imdb.com`). Note that the last two scenarios are based on real-world data. Whereas the CRIME scenario will be used as introductory scenario. the queries demonstrated on the real-world datasets will be inspired by our previous data integration and data cleaning experiences on these data sets. The query complexity will go beyond the toy queries shown in this paper and will include aggregation and negation when applicable. Through these scenarios, we will demonstrate the functionality of all components described in this paper and show how the Nautilus Analyzer successfully allows developers to understand and debug complex SQL queries.

Acknowledgements. This research is partially funded by the Baden Württemberg Stiftung. We thank the students having contributed to the visualization and annotation analyzer components as well as Adriane Chapman and Quoc Trung Tran to have shared their programs with us.

4. REFERENCES

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

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

[3] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2), 2000.

[4] M. Herschel and T. Grust. Transformation lifecycle management with nautilus. In *VLDB Workshop on Quality in Databases (QDB)*, 2011. Long version: `http://www.lri.fr/~herschel/nautilus.pdf`.

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

[6] M. Herschel, M. A. Hernández, and W. C. Tan. Artemis: A system for analyzing missing answers. *PVLDB*, 2(2), 2009.

[7] 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.

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

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

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