Artemis: A System for Analyzing Missing Answers

Melanie Herschel [*]
Universität Tübingen
72076 Tübingen, Germany
melanie.herschel@uni-tuebingen.de

Mauricio A. Hernández [†]
IBM Almaden Research Center
San Jose, CA, 95120, USA
mauricio@almaden.ibm.com

Wang-Chiew Tan [‡]
UC Santa Cruz
Santa Cruz, CA, 95064, USA
wctan@cs.ucsc.edu

1. INTRODUCTION

A central feature of relational database management systems is the ability to define multiple different views over an underlying database schema. Views provide a method of defining access control to the underlying database, since a view exposes a part of the database and hides the rest. Views also provide logical data independence to application programs that access the database. For most cases, the process of specifying the desired views in SQL is typically tedious and error-prone. While numerous tools exist to support developers in debugging program code, we are not aware of any tool that supports developers in verifying the correctness of their views defined in SQL.

Artemis[1] is a system that aims at filling this SQL debugging gap. Our basic vision for Artemis is similar to debuggers for programming languages. As far as we know, the state-of-the-art approach for debugging SQL queries is highly manual and basically requires the database administrator to make a few rounds of study-guess-fix-test cycles (i.e., study the query, guess or pinpoint the problem, fix the query, and test the new query with sample test cases). The goal of Artemis is to provide database administrators and SQL programmers with a facility for understanding and verifying the SQL view specifications more systematically.

In this demo, we showcase the principal novel feature of Artemis. Given a relational database schema and a set of select-project-join-union (SPJU) views defined over it, users can ask why one or more tuples are *not* in the result of the views. This is common in debugging scenarios where oftentimes, one may wonder why the result of a query or view is empty, or why a query did not return certain tuples. In the case of multiple views, one may wonder why, for instance, an employee information is missing from both the employee register and the payroll view. It is also a mechanism for "what-if" analysis, a feature that is commonly sought for in many data analysis applications. As a matter of fact, [2] explored how to explain why a single tuple is not in the result of a single select-project-join (SPJ) query. Artemis extends that work with its ability to generate explanations for a *set* of missing "tuples" over a *set of SPJU queries*. Missing "tuples" may contain labeled nulls representing unknown and potentially shared values across the "tuples". They are encoded into *c-tuples* which are used in deriving explanations for missing tuples (see Sec. 3). Artemis' explanations consist of the set of existing source tuples and the set of missing source tuples that must be added to generate the missing output tuples. Artemis' users can further refine the resulting explanations by requiring Artemis to, e.g., rank the explanations by the number of new tuples they require or by not allowing new tuples to be added to certain views.

The algorithm behind Artemis computes explanations by first encoding the problem into a set of constraints, which is then passed to a constraint solver. The solutions returned by the constraint solver are subsequently filtered and sorted based on user requirements. Artemis' front-end is currently implemented as an Eclipse 3.4 plug-in that extends Eclipse's Data Tool Platform's (DTP) plug-ins.

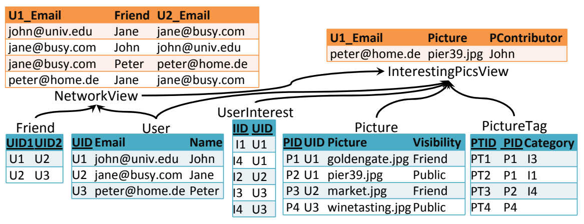
In Sec. 2, we describe the Artemis framework and its extensions. These are then illustrated by a comprehensive example in Sec. 3. We give some details on the underlying algorithm in Sec. 4 before we describe what we plan to demonstrate in Sec. 5.

2. THE ARTEMIS FRAMEWORK

Formally, an Artemis *debugging scenario* is defined by a triple $(\mathcal{Q}, D, E)$. $\mathcal{Q}$ is the set of SPJU queries (or views) that we are debugging. D represents the source database for $\mathcal{Q}$. Let $\mathcal{Q}(D)$ denote the set of tuples in the views. E represents a set of c-tuples that do not exists in $\mathcal{Q}(D)$ and require an explanation. The main goal of Artemis is to generate *explanations* for the results E based on $\mathcal{Q}$ and D. Each explanation describes how existing tuples in D and new c-tuples (not currently in D) are used by $\mathcal{Q}$ to create *all* required tuples in E.

We extend our debugging scenario by allowing users to specify constraints on the returned explanation. Users can select a subset of views $\mathcal{Q}_{im} \subseteq \mathcal{Q}$ and mark them as "immutable views". Intuitively, $\mathcal{Q}_{im}$ corresponds to the queries whose results the user trusts and should not change. Note that the mechanism of $\mathcal{Q}_{im}$ is more general than the trust conditions allowed in [2], where only insertions to source tables can be disallowed through trust conditions placed on those tables. Note, however, that attribute trusts in [2] cannot be captured through queries in $\mathcal{Q}_{im}$, since [2] allows tuples to be inserted into the source table even when certain attributes of the table are trusted. When this constraint is in place, Artemis will not produce explanations that require tuples to be inserted into $\mathcal{Q}_{im}(D)$.

[*]Work done while at IBM Almaden Research Center.
[†]Partially funded by the U.S. Air Force Office for Scientific Research under contract FA9550-07-1-0223.
[‡]Part of the work was done while visiting IBM Almaden Research Center and supported by NSF CAREER Award IIS-0347065 and NSF grant IIS-0430994.
[1]<u>S</u>QL <u>I</u>nspection based on <u>M</u>issing <u>E</u>ntry <u>TRA</u>cing.



NetworkView

U1_Email	Friend	U2_Email
john@univ.edu	Jane	jane@busy.com
jane@busy.com	John	john@univ.edu
jane@busy.com	Peter	peter@home.de
peter@home.de	Jane	jane@busy.com

InterestingPicsView

U1_Email	Picture	PContributor
peter@home.de	pier39.jpg	John

Friend

UID1	UID2
U1	U2
U2	U3

User

UID	Email	Name
U1	john@univ.edu	John
U2	jane@busy.com	Jane
U3	peter@home.de	Peter

UserInterest

IID	UID
I1	U1
I4	U1
I2	U2
I3	U3
I4	U3

Picture

PID	UID	Picture	Visibility
P1	U1	goldengate.jpg	Friend
P2	U1	pier39.jpg	Public
P3	U2	market.jpg	Friend
P4	U3	winetasting.jpg	Public

PictureTag

PTID	PID	Category
PT1	P1	I3
PT2	P1	I1
PT3	P2	I4
PT4	P4	

Figure 1: The PhotoShare database (sample)

```
CREATE VIEW Photoshare.NetworkView AS
SELECT  U1.email AS U1_email,
        U2.name AS Friend, U2.email AS U2_email
FROM    Photoshare.User U1, Photoshare.User U2,
        Photoshare.Friend F
WHERE   U1.uid = F.uid1 and u2.uid = f.uid2
UNION
SELECT  U1.email AS U1_email,
        U2.name AS Friend, U2.email AS U2_email
FROM    Photoshare.User U1, Photoshare.User U2,
        Photoshare.Friend F
WHERE   U1.uid = F.uid2 and u2.uid = f.uid1;

CREATE VIEW Photoshare.InterestingPicsView AS
SELECT  U1.EMAIL AS U1_EMAIL, P.Picture AS Picture,
        U2.Name AS PContributor F
FROM    Photoshare.User U1, Photoshare.User U2,
        Photoshare.PICTURE P, Photoshare.PictureTag PT,
        Photoshare.UserInterest UI
WHERE   U1.UID = UI.UID AND PT.CATEGORY = UI.IID
AND     PT.PID = P.PID AND P.VISIBILITY = 'Public'
AND     U1.UID <> U2.UID AND U2.UID = P.UID
UNION
SELECT  NWV.U1_EMAIL AS U1_EMAIL, P.PICTURE AS Picture,
        NWV.FRIEND AS PContributor
FROM    Photoshare.User U1, Photoshare.User U2,
        Photoshare.PICTURE P, Photoshare.PictureTag PT,
        Photoshare.UserInterest UI, Photoshare.NetworkView NWV
WHERE   NWV.U1_EMAIL = U1.EMAIL AND UI.UID = U1.UID
AND     NWV.U2_EMAIL = U2.EMAIL AND U2.UID = P.UID
AND     PT.PID = P.PID AND PT.CATEGORY = UI.IID
AND     P.VISIBILITY = 'Friend'
```

Figure 2: View definition queries in $\mathcal{Q}$.

Users can also define $\mathcal{Q}_m \subseteq \mathcal{Q}$ as the set of queries for which Artemis must minimize the number of tuples inserted. Note that we require $\mathcal{Q}_m \cap \mathcal{Q}_{im} = \emptyset$. Users can also specify filters on the resulting explanations, for instance, based on the number of tuples inserted. We note that [2] uses a similar concept of trust, but it is less general than the $\mathcal{Q}_{im}$ concept. In addition, [2] does not have a way of specifying $\mathcal{Q}_m$ or other constraints.

3. A SAMPLE TEST CASE STEP-BY-STEP

We illustrate the Artemis framework using a debugging scenario for a social network application. This scenario, called PhotoShare, models a picture sharing application among a network of "friends". Fig. 1 shows an excerpt of the Photoshare database. At the bottom, we depict five source relations. These five tables are D in our debugging scenario. The two top tables depict the result of $\mathcal{Q}(D)$. The corresponding queries in $\mathcal{Q}$ are given in Fig. 2. Arrows from D to $\mathcal{Q}(D)$ in Fig. 1 show which tables are used in the FROM clause of each view definition.

The source schema stores information about PhotoShare users including their friends, pictures taken, and tags about their interests. Each picture can also be tagged as relevant to certain interest.

The *NetworkView* view associates the e-mail addresses of people connected by the *Friends* table. Notice that this association is bi-directional and is computed with the union of two almost identical SPJ queries (one for each direction). The *InterestingPicsView* finds, for each user, a list of shared pictures whose tag matches at most one of the user's interests. For example, peter@home.de appears

	Friend F		User U1			User U2		
	F.UID1	F.UID2	U1.UID	U1.EMAIL*	U1.NAME	U2.UID	U2.EMAIL*	U2.NAME*
1	U1	= U2.UID	U1	john@univ.edu	John	= F.UID2	$friend-email	$friend-name
2	U1	U1	U1	john@univ.edu	John	U1	john@univ.edu	John
3	U1	U3	U1	john@univ.edu	John	U3	peter@home.de	Peter
4	= U2.UID	U1	U1	john@univ.edu	John	= F.UID1	$friend-email	$friend-name
5	U1	U1	U1	john@univ.edu	John	U1	john@univ.edu	John
6	U2	U1	U1	john@univ.edu	John	U2	jane@busy.com	Jane
7	U3	U1	U1	john@univ.edu	John	U3	peter@home.de	Peter

Figure 3: Explanations creating t_1

in this view because the picture pier39.jpg, contributed by John, was tagged as I1, an interest that Peter shares.[2] Given this common database setup, Artemis' users can explore why certain tuples are not in the views. For instance, the query programmer might need to know why john@univ.edu does **not** appear in *InterestingPicsView*. Notice that we can extract several possible high-level explanations by studying the queries. Maybe John is no longer a PhotoShare user, or does not have a declared interest. Perhaps none of the other users is sharing a photo tagged with John's interest. Or, even if they have such a photo, the photo is not visible to John.

To explain this missing value, the user enters a new c-tuple

$$t_2 = (\text{'john@univ.edu'}, \$\text{picture}, \$\text{friend-name})$$

into *InterestingPicsView*. Further, assume the user wants to ensure 'john@univ.edu' is a friend of the person contributing the picture. This is easily done by entering a new c-tuple

$$t_1 = (\text{'john@univ.edu'}, \$\text{friend-name}, \$\text{friend-email})$$

into *NetworkView*. The c-tuples contain constant values and labeled nulls, denoted by a \$ sign followed by the name of the null[3]. Notice that the null friend-name is present in both t_1 and t_2, meaning that both tuples share the same (unknown) value.

Fig. 3 shows some possible explanations for t_1. Each explanation (i.e., row) contains data from a single tuple of each source table in the query. In the case of $t_1 \in$ *InterestingPicsView*, each explanation has a tuple from *Friend* and two tuples from *User*. Notice that some of these tuples (those with the lighter background) already exist on the source tables and others (those with the darker background) are c-tuples that need to exist for t_1 to exist. Consider, for example the last explanation in Fig. 3 (row 7). To create t_1 using that explanations, a tuple (U3, U1) must be inserted into *Friends*. That tuple will join with the existing tuples in *User* shown in the same explanation row. Similarly, Fig. 4 shows two alternative explanations to produce t_2 (we use '?' to denote any possible values).

To explain t_1 and t_2 at the same time, we compute the cross product of all explanations for t_1 and t_2. As we do this, we ensure that conditions that appear in the individual explanations appear in the combined explanation. For example, shared labeled nulls like friend-name are assigned the same value.

Artemis produced 469 such explanations for this debugging scenario (the cross product of the 7 explanations of t_1 and the 67 explanations of t_2). All explanations satisfy source constraints (key, unique, foreign). It is possible to trivially extend the algorithm in [2] to support multiple SPJU queries and explain multiple tuples. However, since [2] does not consider these constraints, they return 9900 explanations (25 for t_1 and 396 for t_2).

Artemis provides several ways to reduce the number of explanations. Our user can, for instance, add *User* to $\mathcal{Q}_{im}$ to signal

[2]The *InterestingPicsView* query is a union between shared pictures that are visible to everybody and pictures that are only visible to friends.

[3]As we shall see in Sec. 4, **c**-tuples also include a **c**ondition that determines if the tuple exists in an instance.

	Picture P				PictureTag PT			UserInterest UI		User U1			User U2			NetworkView NWV		
	P.PID	P.UID	P.PICTURE*	P.VISIBILITY	PT.ID	PT.PID	PT.CATEGORY	UI.UID	UI.IID	U1.UID	U1_EMAIL*	U1.NAME	U2.UID	U2.EMAIL	U2NAME*	NWV.U1_EMAIL	NWV.FRIEND*	NWV.U2_EMAIL
1	= PT.PID	= U2.ID	$picture	Public	?	= P.PID	= UI.IID	= U1.ID	= PT.CATEGORY	U1	john@...	John	= P.UID & ≠ U1.ID	?	$friend-name			
2	= PT.PID	= U2.ID	$picture	Friend	?	= P.PID	= UI.IID	= U1.ID	= PT.CATEGORY	U1	john@...	John	U1	john@...	John	john@univ.edu	$friend-name	= U2.EMAIL

Figure 4: Explanations creating t_2

that no changes can be made to *User*. This removes from consideration any explanation that inserts a c-tuple into *User*. Another possibility is to restrict or minimize the "side-effects" an explanation creates. For example, consider what happens if we execute the explanation for t_1 in row 3 (Fig. 3). That explanation requires adding a tuple (U1, U3) to *Friend*. However, that tuple will create a new tuple (peter@home.de, goldegate.jpg, John) in *InterestingPicsView*. Since this tuple is not t_2, is it considered a "side-effect". To avoid these cases, users can add *InterestingPicsView* to $\mathcal{Q}_m$ and Artemis will only produce explanations that minimize changes to that view (in this case, only t_2). With these simple constraints in place (namely, $\mathcal{Q}_{im} = \{$*User*, *UserInterest*$\}$, and $\mathcal{Q}_m = \{$*InterestingPicsView*$\}$), Artemis returns a total of 52 explanations: 4 explanations for t_1 and 13 explanations for t_2.

4. A GLIMPSE BEHIND THE SCENES

The Artemis algorithm determines the provenance of missing tuples specified in E. Various notions of database provenance exist. This work is largely inspired by [2], where data provenance is extended to explain non-existing data. That work is, however, limited to a single SPJ query in Q and a single tuple in E.

The basic Artemis algorithm consists of five major steps, which we briefly outline here.

Step 1: Compute generic witness. Using $\mathcal{Q}$ and E, Artemis computes a *generic witness* that produces all c-tuples in E. The generic witness is a set of source tuple "patterns" that match the tuples in the generated explanations. For example, if we assume we only want to explain t_1 (see Fig. 3), Artemis generates a generic witness with the following two patterns:

$$\begin{aligned} \text{Pattern}_1 &= Friend(uid1, uid2), U1(uid1, \text{john@univ.edu}, n1), \\ &\quad U2(uid2, \text{\$friend-email}, \text{\$friend-name}) \\ \text{Pattern}_2 &= Friend(uid2, uid1), U1(uid1, \text{john@univ.edu}, n1), \\ &\quad U2(uid2, \text{\$friend-email}, \text{\$friend-name}) \end{aligned}$$

Notice that Pattern$_1$ matches rows 1-3 in Fig. 3 and Pattern$_2$ matches rows 4-7. A nice feature of the generic witness is that it serves as a summary of the actual explanations (and can be used to hide explanation tuples and declutter the display).

For each pattern in the generic witness, Artemis executes Steps 2, 3 and 4.

Step 2: Create c-tables for D. In this step, Artemis creates conditional tables (c-tables) for the source data in D. A c-tuple in a c-table includes labeled nulls and conditions. The semantics of a tuple in a c-table are that a tuple only exists if its condition holds [3]. When generating c-tables for D, tuples already in D are assigned the TRUE condition. New tuples, those "generated" by the witness pattern, are represented using labeled nulls and conditions. The condition of a such tuples is the conjunction of all the conditions in the witness pattern plus conditions implied by key, unique, foreign key constraints. Fig. 5 shows the result of Step 2 for P$attern_1$ using our debugging scenario. In a sense, [2] implicitly performs this step, but only for the special case of a singe SPJ query and a single missing tuple. This is the last step for which we can make analogies to [2] as all remaining steps relate to the more general case Artemis considers.

Step 3: Execute $\mathcal{Q}$ over the c-tables. Let D_c be the c-table version of the source database D produced by Step 2. Artemis computes

User

UID	Email	Name	Condition
U1	john@...	John	True
U2	jane@...	Jane	True
U3	peter@...	Peter	True
uid1	john@...	n1	uid1 & email unique
uid2	n2	e2	uid1 & email unique & n2 = $friend-name & e2 = friend-email

Friend

UID1	UID2	Condition
U1	U2	True
U2	U3	True
fuid1	fuid2	(fuid1,fuid2) unique, foreign key dependencies for fuid1 & fuid2 hold

Figure 5: Example for generated c-tables for D

$\mathcal{Q}(D_c)$ using the algorithms for query execution over c-tables [3]. The result are views with the same schema as originally plus a condition attached to each tuple. In these views we distinguish three types of tuples: (i) tuples that are present in the original view (those have a TRUE condition), (ii) tuples that potentially match tuples in E, and (iii) the remaining tuples, which are potential side effects. In Fig. 6, we only show tuples of type (ii) (those with a check mark) and type (iii) (those with an 'x') in the c-table version of the *NetworkView*. The conditions enforcing key, unique, and foreign key constraints are omitted for brevity. In the following, we will refer to the different conditions as c_1 through c_7, where c_1 is the condition of the first tuple, c_2 the condition of the second tuple and so on.

Step 4: Compute explanations. Next, Artemis determines every possible way of combining tuples of type (ii) such that each of the n tuples in E is generated exactly once. For each combination, a condition $C_i = c_{t1} \wedge ... \wedge c_{tn}$ is formulated that corresponds to the conjunction of the conditions attached to each of the n tuples. All other tuples of type (ii) or (iii) are considered side-effects and we append to C_i the constraint that none (or the minimal number) of these tuples should exist. In our example, we have $n = 1$ tuples in E, and 2 possibilities to create it. Assuming that no side-effects on *NetworkView* are desired, the generated conditions are

$$\begin{aligned} C_1 &= c_1 \wedge \neg(c_2 \vee c_3 \vee c_4 \vee c_5 \vee c_6 \vee c_7) \\ C_6 &= c_6 \wedge \neg(c_1 \vee c_2 \vee c_3 \vee c_4 \vee c_5 \vee c_7) \end{aligned}$$

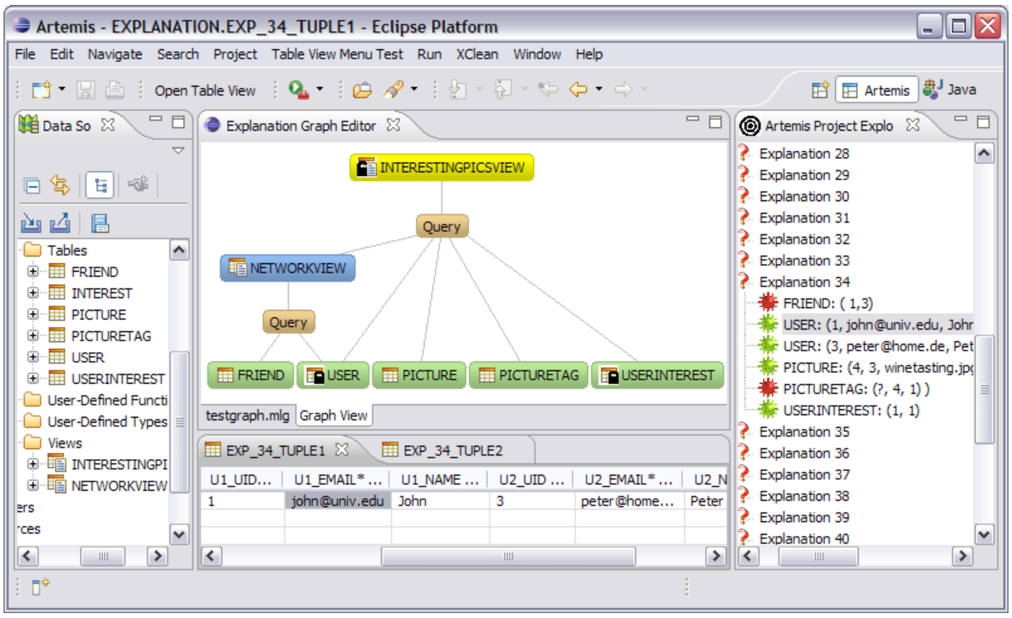
We pass each C_i to a constraint solver [1] and if it determines that the problem is satisfiable, we generate the corresponding explanation. Of the constraints shown above, only C_1 is satisfiable and corresponds to the insertion of (U1,U1) into the *Friend* relation. C_6 is not satisfiable because any solution that creates (john@univ.edu, Peter, peter@home.de) in *NetworkView* creates (peter@home.de, John, john@univ.edu) as a side-effect, i.e., $c_6 \wedge \neg c_3$ is not satisfiable and hence C_6 is not satisfiable.

Step 5: Prune and output explanations. In the final step, Artemis unions the explanations returned for each pattern of the generic witness (Step 2 through Step 4 iterated over these). Artemis prunes redundant solutions if necessary and applies any filters that have not been considered so far. It also sorts the remaining explanations if necessary before returning them.

Steps 1 through 3 resemble the well-studied problems of view update and view maintenance, and the definition of these steps of the Artemis algorithm was inspired by the work on view maintenance and view update using c-tables presented in [5, 6]. Runtime performance of the constraint solver is acceptable. For the small example in this paper, the solver takes 2 seconds. With larger do-

	U1_EMAIL	Friend	U2_EMAIL	Condition
✓	john@univ.edu	John	john@univ.edu	c_1 = (F_UID1 = U1_UID) & (F_UID2 = U2_UID) & (U1_UID = U1) & (U2_UID = U1) & $friend-name = John & $friend-email = john@univ.edu
✗	jane@busy.com	John	john@univ.edu	c_2 = (F_UID1 = U1_UID) & (F_UID2 = U2_UID) & (U1_UID = U2) & (U2_UID = U1) & $friend-name = John & $friend-email = john@univ.edu
✗	peter@home.de	John	john@univ.edu	c_3 = (F_UID1 = U1_UID) & (F_UID2 = U2_UID) & (U1_UID = U3) & (U2_UID = U1) & $friend-name = John & $friend-email = john@univ.edu
✗	jane@busy.com	Jane	jane@busy.com	c_4 = (F_UID1 = U1_UID) & (F_UID2 = U2_UID) & (U1_UID = U2) & (U2_UID = U2) & $friend-name = Jane & $friend-email = jane@busy.com
✗	peter@home.de	Jane	jane@busy.com	c_5 = (F_UID1 = U1_UID) & (F_UID2 = U2_UID) & (U1_UID = U3) & (U2_UID = U2) & $friend-name = Jane & $friend-email = jane@busy.com
✓	john@univ.edu	Peter	peter@home.de	c_6 = (F_UID1 = U1_UID) & (F_UID2 = U2_UID) & (U1_UID = U1) & (U2_UID = U3) & $friend-name = Peter & $friend-email = peter@home.de
✗	peter@home.de	Peter	peter@home.de	c_7 = (F_UID1 = U1_UID) & (F_UID2 = U2_UID) & (U1_UID = U3) & (U2_UID = U3) & $friend-name = Peter & $friend-email = peter@home.de

Figure 6: Excerpt of the c-table for NetworkView



U1_UID...	U1_EMAIL* ...	U1_NAME ...	U2_UID ...	U2_EMAIL* ...	U2_N
1	john@univ.edu	John	3	peter@home...	Peter

Explanation 28
Explanation 29
Explanation 30
Explanation 31
Explanation 32
Explanation 33
Explanation 34
FRIEND: (1,3)
USER: (1, john@univ.edu, John
USER: (3, peter@home.de, Pet
PICTURE: (4, 3, winetasting.jpg
PICTURETAG: (?, 4, 1))
USERINTEREST: (1, 1)
Explanation 35
Explanation 36
Explanation 37
Explanation 38
Explanation 39
Explanation 40

Figure 7: Artemis in action

mains, e.g., with 100 distinct values per attribute, average runtime is below 20 seconds. Improving and optimizing our runtime is a major direction of further research.

5. THE DEMONSTRATION

We implemented an optimized version of the Artemis algorithm in Java. The implementation requires a constraint solver and the current implementation uses Minion 0.7.0 [1]. The graphical user interface was created extending Eclipse plugins and allows users to specify parameters of Artemis debugging scenarios, run debugging scenarios, and explore the returned explanations. The Artemis plugin comes with many features known in the world of Eclipse, including its own Perspective, Views, Editors, and Project based resource organization.

Fig. 7 shows a screenshot of Artemis in action. The Data Source Explorer on the left allows users to specify $\mathcal{Q}$, which is an extension of the explorer provided by Eclipse's Data Tools Platform The Graph View in the center summarizes $\mathcal{Q}$ and D in a similar way as Fig. 1 does for our sample scenario. The Explanation Navigator on the right shows summaries of each explanation and the Explanation Detail View at the bottom shows the details. Note that the screenshot exactly represents the sample scenario discussed in this paper. Our current implementation supports both DB2 and Derby as source databases storing D.

At the conference we will demonstrate Artemis using several debugging scenarios for at least two use cases. The first use case, named PhotoShare includes the toy examples used throughout this paper as well as more real-life debugging scenarios for an application that allows users to connect to each other and share, tag, or comment pictures. The second use case is based on the TPC-H schema and data. However, since Artemis does not support aggregations, only a limited number of TPC-H queries will be used[4]. We will show attendees how Artemis helps understanding and debugging these SQL queries, and we will discuss the details Artemis' algorithm, limitations, and future directions.

6. REFERENCES

[1] I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006.

[2] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. In *PVLDB*, pages 736–747, 2008.

[3] T. Imieliński and J. Witold Lipski. Incomplete information in relational databases. *JACM*, 31(4):761–791, 1984.

[4] J. Lechtenbörger, H. Shu, and G. Vossen. Aggregate queries over conditional tables. *J. Intell. Inf. Syst.*, 19(3):343–362, 2002.

[5] H. Shu. View maintenance using conditional tables. In *DOOD*, pages 67–84, 1997.

[6] H. Shu. Using constraint satisfaction for view update. *J. Intell. Inf. Syst.*, 15(2):147–173, 2000.

[4]Aggregation can be supported through the algorithm described in [4] but some unresolved issues remain at present.