

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN



Diploma Thesis

Usability of the Nautilus-Analyzer with Visualization and User-interaction

Bastian Brodbeck
bastianbrodbeck@mac.com

SS 2010

Mentoring: Dr. Melanie Herschel

Contents

1. Introduction	3
1.1. Goals	4
2. Nautilus	4
2.1. Introduction to the Algorithm	7
2.2. Pattern as Core-Structure	9
2.3. Incremental Computation	11
2.4. Reusable Data	12
3. The Visualization	16
3.1. The GUI Elements	17
3.1.1. Graph Layouts	23
3.2. GUI Interactions	24
3.3. Interactions with the Algorithm	24
3.4. Use-Case Scenario	25
4. Tests and Evaluation	29
4.1. The Debugging Scenarios and Test Platform	29
4.2. Tests and Evaluation	30
5. Disadvantages, Problems and possible Solutions	32
5.1. Graph Implementation	33
6. Conclusion	34
A. Database	36
A.1. Photoshare	36

1. Introduction

A lot of Applications, no matter what type, i.e. desktop-, mobile- or web-based, are attached to a database holding all necessary data. The data can be stored and it can be aggregated and combined to get new informations. A standardized language, the *Structured Query Language*, SQL, was created for relational databases to achieve such tasks. A database request — or query — defines the tables and columns of a certain database needed for a task and how to combined them.

When working with SQL one often wonders why certain tuples are missing in the result of such a query. The returned result set might be empty or missing an important tuple the programmer expected to see. Finding the cause of the missing results is often complex and time consuming.

There are two main reasons why one or more tuples might be missing. Tuples are mostly the result of several source tables within a database combined by certain rules given by a query. The query defines with an relational algebra the source tables and how to combine the matching tuples of these tables. One reason why a tuple does not appear in the result set is that the rules defined by the query might disqualify a certain combination, or a join of two source tuples. The more complex a query gets, the more likely it is that a certain join disqualifies a tuple. To analyze such a faulty query takes time — depending on the complexity of the query, the complexity of the database schema and the size of the data stored within the database. The answer why a tuple is missing for such a scenario is called a *query-based explanation*.

The second reason is a missing tuple within the stored database. What if a certain join wants to combine, lets say, an employee and the department he is working in, but the tuple of the referenced department is missing? The query works perfectly and the programmer will not find a way to fix the result set by adjusting the query. This is called an *instance-based explanation* of a missing tuple. To find such an error we too need time and the knowledge of the database schema and its stored data.

Nowadays a programmer has the luxury of having debugging and optimization tools for a wide variety of problems for several programming languages or markup languages but currently there are no tools for computer assisted debugging for either query-based or instance-based explanations available.

Two Teams tried to close this gap of missing debugging tools by finding an algorithm that can identify the missing or faulty information in a sql query — or database instance respectively.

A. Chapman and H. V. Jagadish published 2009 an algorithm to get query-based explanations for missing tuples in their paper [4].

In 2010 M. Herschel and M. A. Hernández published an algorithm [5] that addresses the instance-based explanations instead.

1.1. Goals

This work is based upon the second algorithm called Artemis and its implementation as an Eclipse Plugin [1]. The first goal was to extend the eclipse plugin of Artemis demoed in [6] with a better graphical user interface. The interface should visualize the explanations and their meanings and allow the user to influence the algorithm while it is calculating in the background.

To achieve this I introduce two new graphs for visualization, selection and filtering as well as for interacting with the algorithm. In Addition I modified the parts of the interface listing the explanations to reflect the changes in the algorithm and added a new control to "steer" the direction of the algorithm. Se more in Section 3.

The second goal was to identify possible changes in the algorithm to get speed-up on the whole.

First I restructured the algorithm to minimize the needed calculations, then I optimized the order certain explanations or parts of them are calculated.

While the restructuring was intended to effect on the runtime of the algorithm, the ordering was supposed to affects the impression a user has of the speed i.e. the time the first tuples are finished.

The algorithm was also adapted to possible user interactions. It is now possible to stop a certain calculation and resume it later.

All this was possible because of the restructuring of the algorithm core without changing the basic idea behind it. See more details in Section 2.1.

My work was split in three major parts

1. **Theroy:** what changes might have a positive effect
2. **Implementation:** can the theory be added to the algorithm
3. **Testing:** is the effect as anticipated

In this theses I will combine the theory with the implementation by showing the idea and where necessary pointing to the implementation.

2. Nautilus

Let me introduce an example for the typical workload for this tool before we go deeper into Nautilus (formally known as Artemis).

2. Nautilus

Example 1: Say we have an online platform for gamers in the internet. The gamers can buy games, join groups to certain games and befriend each other. Each group is linked to one or more games and a user only can join a group if he has bought at least one of the games. Friends can see the games each other bought and which groups the other has joined. In addition a user can state in what type of games he is interested.

Now say, such a platform was online for a while and the users were quite active, so our database is full of informations and we want to take a look at those informations to see which games need more advertising. Further say we want to advertise certain games whose type a user is interested. We assume a game G is known to a user U if he has joined a group corresponding G or if he has at least one friend who has bought G .

We have the following three criteria:

1. U has interest in games like G
2. U is not part of a group corresponding G
3. U has no friends owning G

We can formulate those criteria in SQL to find all matching games. Now if we are debugging a database to test our queries we know what a result set should look like. In addition we might wonder why for user U_x the game G_y is missing in the result set.

With the artemis algorithm we are able to formulate a request that explains why the tuple for game G_y is missing and more important what do we need to add in our database to change that. In our Example we cannot force a user to change his interests to get a certain game. This is already a criteria for the SQL query but not for artemis because we are searching for missing database entries. One type of explanation for a missing tuple might be: "The user has no interest in such games, so we need to add it." Another explanation might be: "The user needs to befriend user U_z "; again we cannot force our user to do that. Artemis allows us to ignore such answers. We can add to our request:

1. Don't change the interests
2. Don't change the friends

What we can do is suggest to a user to join a certain group instead of advertising the game directly hoping he notices the game through that group; we even can generate new groups to connect more games and users that way.

With Example 1 I showed a scenario were we can use the artemis algorithm not only for debugging but also for analyzing our database to achieve a certain goal. The input for the artemis algorithm and various parameters as seen in the example is called a *debugging scenario* which is defined by [5] as the following tuple:

Definition 1: Debugging Scenario

A debugging scenario \mathcal{S} is a 5-tuple $\langle Q, Q_{im}, Q_m, D, E \rangle$. Q is the set of queries we are debugging. D represents the source database for Q and we write $Q(D)$ to address a set of result tuples for Q executed on D . With E we represent a set of c-tuples that encode missing tuples that are not element of $Q(D)$ and we need an explanation for.

Assume there exists a view definition V_i for each relation $R_i \in D$ such that $V_i(X) : -R_i(X)$ (i.e. each view mirrors a source relation). The set containing all these identity views is called Q_v .

Q_{im} represents the set of immutable queries with $Q_{im} \subset Q \cup Q_v$. An explanation is not allowed to have any side-effects on an immutable query. $Q_m \subset Q \cup Q_v$ is a set of queries where minimal side-effects are allowed on.

Further, $Q_{im} \cap Q_m = \emptyset$

Lets translate this definition to Example 1. The database D is our database containing all the tables and their tuples for our platform. Q contains our SQL query we get from our three criteria. We acknowledged that we cannot force our users to befriend certain other users or that they change their interests, so the tables or views representing those relations in our database are element of Q_{im} . Q_m is empty for know.

A SQL query can be any query of type SPJUA, that is a query that can contain Selections, Projection, Joins, Unions and Aggregations. In the following we will only discuss SPJU queries. Before we look into E we need another definition:

Definition 2: Conditional Tuple

A conditional tuple (c-tuple) is a tuple $\langle a_1, a_2, \dots, a_n, cond \rangle$ such that every value a_i is either a constant or a labeled null. The attribute $cond$ is a Boolean expression. C-tuples populate conditional tables (c-tables) whose relational extension contains all c-tuples whose condition evaluates to true.

The advantage of conditional tuples to formulate missing tuples is - that lets say we have a game not suitable for persons under the age of 18 - we can easily filter unnecessary explanations beforehand by adding a constraint. If we have a view *UserInterests* in our Example 1 that shows us the username, his age and a certain interest we could formulate a c-tuple like:

$$t = UserInterest(\$name, \$age, 'Shooter', \$age \geq 18)$$

The tuple t contains two labeled nulls ($\$name$ and $\$age$), a constant ('Shooter') and the condition, the last value, that $\$age$ should be at least 18. Now we know how the tuple $t \in E$ for a debugging scenario look like.

To understand the algorithm we need to define what an explanation exactly is; for that we need the following definition:

Definition 3: Pattern

2. Nautilus

A pattern P_{ij} is a finite set of c-tuples that if inserted into D generates a subset of c-tuples in E for a certain query $Q_i \in Q$; further generates $P_i = \cup_j P_{ij}$ **all** c-tuples in E for Q_i .

Definition 4: Witness

A witness w is a set of pattern so that it contains exactly one pattern for each query $Q_i \in Q$

Note: Don't confound this *pattern* with the *explanation pattern* defined in [5]; the witness definition is the equivalent to an explanation pattern.

Now we have everything we need to finally define an explanation.

Definition 5: Explanation

A sub-explanation is a c-tuple that matches a pattern P_{ij} . An explanation e is a not empty set of sub-explanations. The c-tuples from the sub-explanations matches one witness.

Definition 6: Generic Witness

The generic witness W for a scenario $\cap S$ is a set of witnesses such that any explanation e in E matches at least one witness.

2.1. Introduction to the Algorithm

Like Artemis, the renamed and extended Nautilus-Algorithm has four major steps. In the following I will summarize the basic ideas behind these four steps and point out the major differences between Artemis as in [5] and my Nautilus algorithm. In the following subsections we will look deeper into the changes and their motivations.

Step 1: compute generic witness

If Q only contains one SPJU query then the unions of conjunctive queries each represents the base in combination with tuple t for a pattern as described in Definition 3 and each pattern is a witness for our debugging scenario \mathcal{S} . But Artemis accepts more than one query and more than one tuple as input. Each tuple $t \in E$ occurs by definition simultaneously and therefore our generic witness needs to generate **all** explanations for **all** tuples.

First we determine the generic witness for each tuple $t_i \in E, 1 \leq i \leq n$ and then we use the cross-product to combine those generic witnesses to the generic witness of \mathcal{S} .

$$W = W_1 \times W_2 \times \dots \times W_n$$

2. Nautilus

Example 2:

$$\begin{aligned}\text{Query } Q_1 : q_{11} \cup q_{12} &\rightarrow \text{Patterns: } p_{11}, p_{12} \\ \text{Query } Q_2 : q_{21} \cup q_{22} \cup q_{23} &\rightarrow \text{Patterns: } p_{21}, p_{22}, p_{23} \\ \text{Query } Q_3 : q_{31} &\rightarrow \text{Patterns: } p_{31}\end{aligned}$$

Generic Witness:

$$\{p_{11}, p_{12}\} \times \{p_{21}, p_{22}, p_{23}\} \times \{p_{31}\} = \left\{ \{p_{11}, p_{21}, p_{31}\}, \{p_{11}, p_{22}, p_{31}\}, \{p_{11}, p_{23}, p_{31}\}, \right. \\ \left. \{p_{12}, p_{21}, p_{31}\}, \{p_{12}, p_{22}, p_{31}\}, \{p_{12}, p_{23}, p_{31}\} \right\}$$

Example 2 shows us, how we obtain the generic witness for three SPJU queries — we assume there is one tuple for each Query in set E .

Query Q_1 has two subqueries q_{11} and q_{12} and the algorithm computes a pattern for each. Query Q_2 has three subqueries and therefore three patterns and Q_3 has one pattern.

The generic witness for one of these queries alone would be the corresponding set of patterns, each pattern being one witness. If we have all three queries in our debugging scenario then we get the cross-product as seen in the Example. We have three queries so each witness consists of three patterns, one pattern for each query, and we get $2 \cdot 3 \cdot 1 = 6$ witnesses.

Step 2: create c-tables

First every source table of our database D is converted to c-tables by adding the condition 'true'; I will refer to those as D_c . After this conversion the algorithm sorts the witnesses and iterates through each pattern of each witness executing the rest of Step 2 as well as Step 3 and 4.

The c-tables for the current pattern are added to the c-tables D_c before executing step three.

Step 3: execute Q over c-tables

The algorithm executes $Q(D_c)$. As return we get a set of possible explanations — satisfiable as well as unsatisfiable.

Step 4: compute explanations

First the algorithm searches for actual matches for each c -tuple of the currently calculated pattern in the c -tuple of the explanation result set of step 3. Two c -tuples match if the constants match, labeled nulls are unified and the condition is satisfiable.

Next a constraint solver determines if all constraints in the query / pattern, in t and in Q_m , Q_{im} are satisfiable.

Note: For further details on Step 2-4 consult [5], since there are little to no changes made in these steps.

2.2. Pattern as Core-Structure

In most debugging scenarios the size of set Q — the queries to debug — will be > 1 and therefore a witness will contain several pattern — to be exact the size of a witness is $|w| = |Q|$ for each witness $w \in W$.

While the old Artemis Algorithm was more or less query-tuple-oriented, the new algorithm is, as suggested by [5], more Pattern-oriented. In the past, the algorithm formed pairs of an query and a tuple; if one wants to find an answer for multiple tuples, there would be a redundant number for each query and thus each query would be analyzed and calculated several times, although the query itself does not change its behavior with each tuple. A Pattern represents one query, and all its informations, including all tuples to be inserted.

In Artemis a witness — or an explanation pattern as it is called in [5] — is calculated as one entity for each tuple we want an explanation. In the new pattern oriented way the explanations a pattern produces are only part of an complete explanation a witness produces. If the queries in our query set Q of the debugging scenario \mathcal{S} are unions of sub-queries, than a pattern is part of several witnesses, we can therefore reuse the explanations of the pattern for each witness.

This way, we reduce the overall cost to calculate every explanation. The explanations for a witness is the cartesian product of the explanations for each pattern in the witness. Its the same technique we use for generating the generic witness set, seen in Example 2.

As explained earlier the algorithm finds generic witnesses composed of patterns. For each query in our workload, there is exactly one pattern in each generic witness. After identifying the generic witnesses the algorithm iterates over the set of generic witnesses to find the matching explanations.

Let's say we have three queries as seen in Example 2. Nautilus generates as expected six generic witnesses, each containing three patterns. Now the algorithm iterates over

2. Nautilus

the set. Be the first generic witness $\{p_{11}, p_{21}, p_{31}\}$ and the second one $\{p_{12}, p_{22}, p_{31}\}$. As we can see only pattern p_{31} is the same in both witnesses. Before we get the first explanation of generic witness two, we need to calculate pattern p_{12} completely and we have to begin calculation pattern p_{22} . With the first correct explanation for pattern p_{22} we get the first set of explanations for this generic witness as a result of the cartesian product of all explanation of all three patterns.

If we have a large set of generic witnesses and/or each generic witness contains a large numbers of patterns and we have no ordering of our witness-set, then there are generic witnesses disjunct to each other and the probability that we have to calculate these two one after the other is > 0 .

If we are only interested to get all possible explanations at some point, then this is efficiently enough, but with the goal to visualize the explanations "as they come" and to interact while the algorithm is still running this is not acceptable. Further, if a user is waiting, then this approach might take more time to get the first results than necessary. To avoid those unneeded time periods we sort the generic witnesses before calculating the explanations. Since the generic witnesses are the result of a cartesian product we know, that there is always a generic witness containing exactly the same patterns except one. With this knowledge we can sort the witness set following one simple rule:

1. The next generic witness contains **at most** one not yet calculated pattern.

Of course the first generic witness has to break this rule, but we can sort the following generic witnesses this way. Let's try it with our witness-set from Example 2.

Example 3: The ordering might look like this:

$$\{\{p_{11}, p_{21}, p_{31}\}, \{p_{11}, p_{22}, p_{31}\}, \{p_{11}, p_{23}, p_{31}\}, \{p_{12}, p_{21}, p_{31}\}, \{p_{12}, p_{22}, p_{31}\}, \{p_{12}, p_{23}, p_{31}\}\}$$

We begin with $\{p_{11}, p_{21}, p_{31}\}$ and next we just need to calculate pattern p_{22} followed by p_{23} .

Of course the overall speed does not improve, but we minimize the time before we get the first explanation for each generic witness following the first one. This improvement reduces the time a user has to wait before he can interact with the results in the visualization and therefore (might) give the user the feeling of a fast algorithm overall.

There is another advantage of an ordering of patterns. A user might not need all explanations for each witness to find a solution for it and might want to see other witnesses before the current witness is finished calculating. If a user selects another witness we reorder our patterns according to this witness to get a new optimal ordering while utilizing the already finished patterns.

In Example 3 above the first witness selected was $\{p_{11}, p_{21}, p_{31}\}$. Lets say we select $\{p_{12}, p_{23}, p_{31}\}$ instead. The new ordering might look like:

$$\{\{p_{12}, p_{23}, p_{31}\}, \{p_{12}, p_{21}, p_{31}\}, \{p_{12}, p_{22}, p_{31}\}, \{p_{11}, p_{23}, p_{31}\}\{p_{11}, p_{21}, p_{31}\}, \{p_{11}, p_{22}, p_{31}\}\}$$

2. Nautilus

As soon as p_{12} and p_{23} are calculated — both not part of the original witness — we have everything calculated for the second witness in the new order $\{p_{12}, p_{21}, p_{31}\}$.

The advantage is, that the user does not have to wait before we reach the new witness in the original order. We reorder because we assume that the user wants to see the witness neighborhood, that is similar witnesses, more often than complete different witnesses. Even if the user chooses to see a disjoint witness next, it would be near impossible to order the witnesses incorporating such random selections for a large generic witness set.

2.3. Incremental Computation

Sometimes it might be useful to stop a pattern mid-calculation, for example as seen above, if the user switches a witness. For this we need certain pattern states that are save stop and resume, because if we can stop a pattern it would be nice to resume where we left.

Of course it is not possible for the algorithm to stop a pattern everywhere. Therefore I added several exit-points in the pattern-structure where the algorithm checks for any "pattern-switch-requests". I added these exit points after every major calculation step and after each iteration where the constraint-solver checks a possible explanation if it is satisfiable or not.

Figure 1 shows a graph representing the several states a pattern has. Each node represents one state of the pattern. A state is stable configuration of our pattern and allows us to exit the pattern calculation. If we resume calculating a pattern we re-enter at the last stable state we achieved.

To achieve this function in the fourth step of the algorithm, the Solver Step, there is a little more overhead in Nautilus needed than in Artemis. Artemis sent for each *explanation pattern* the conditionalized query as request to the database, searched the result set for matching tuples and then iterated trough this set of matching tuples and sent a request for each one to the constraint solver.

On the whole Nautilus does the same, but there are two major differences. First: after receiving the database answer, Nautilus too checks for matching tuples and saves those as possible explanations within the patter resulting in a higher memory usage. At this point Nautilus can stop this pattern or resume the pattern at this point. All data we need is stored within the pattern.

Afterwards Nautilus iterated trough the set of matching tuples and sends for each one a request to the constraint solver. After receiving the answer, the possible explanation is marked as satisfiable or not, and a 'matching tuple' pointer is incremented. After each incrementation Nautilus can stop or resume this pattern, because the pointer shows to the next calculated-to-be tuple.

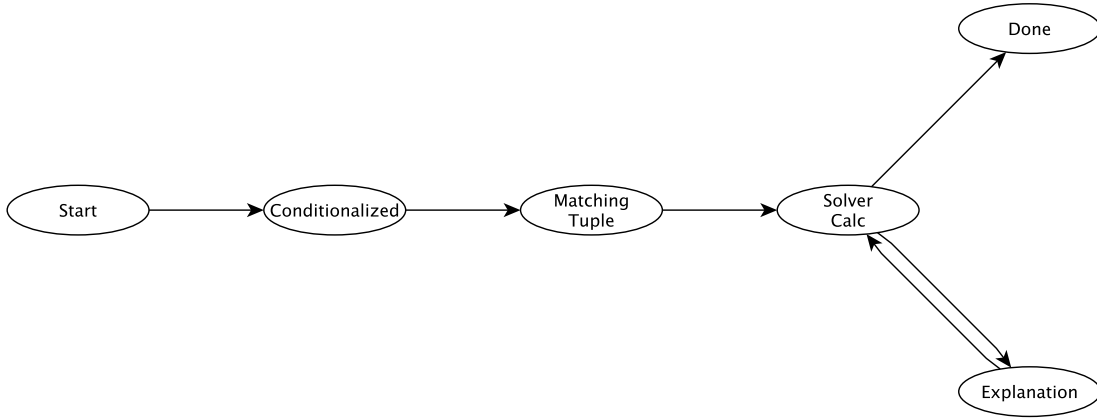


Figure 1: basic states of a pattern

2.4. Reusable Data

When working with large sets of data and complex calculations one might want an option to save the result of the calculations. Before, there was no possibility to save the generated explanations of a certain debugging scenario \mathcal{S} in Artemis.

In Nautilus it is possible to use the last saved calculations of a certain debugging scenario \mathcal{S} and save the current ones for future usage (called History \mathcal{S}_H). There are several restrictions at the moment that can be addressed in the future.

1. Only one calculation state can be saved for each scenario. This history \mathcal{S}_H gets overwritten after a new calculation.
2. The Sets Q , E , Q_m and Q_{im} must be the same in the saved history \mathcal{S}_H as in the current scenario \mathcal{S} .
3. A debugging scenario \mathcal{S} either matches a history \mathcal{S}_H completely or \mathcal{S}_H is not used at all. Here is the biggest possibility for optimization in the future. If some queries in the history \mathcal{S}_H and their patterns are also used in the current debugging scenario \mathcal{S} with the same tuple(s) than it is possible to reuse the old results.
4. The history is saved in the same path as the project files for a debugging scenario within Eclipse. This means if there is an identical debugging scenario \mathcal{S} in a different project, Nautilus does not detect the matching history.

The history of a scenario can not only be used to revisit an old complete calculation but to resume an aborted one. The history is treated the same way than newly generated data and therefore uses all the same techniques like resuming an unfinished pattern.

2. Nautilus

This is especially useful for large scenarios.

To achieve this feature of resuming a history I needed to change the order of the first steps of the algorithm. In Artemis the first step was computing the generic witness set and the second step began with calculating and conditionalizing Q_m and Q_{im} followed by the conditionalization of the source tables. Now the ordering is the following (numbered by their old position):

- 2a. prepare Q_m and Q_{im} then check if the history is compatible
1. compute generic witness set in case the history is non-existent or does not fit or else load the history
- 2b. create c-tables of the source tables

The history is composed of two files; one for the informations about the saved debugging scenario and a second one for all basic informations about the patterns of the scenario.

History: The History contains information about the debugging scenario

- Queries Q
- Tuples E
- Immutable Queries Q_{im}
- queries with minimal side-effect Q_m
- Generic Witness Set W containing sets of witnesses / patterns
- a data structure grouping patterns to their query / database view

Pattern: This file contains the following informations for all patterns

- identification of the pattern, a 64-bit integer
- the query / database view name the pattern is part of
- the query itself as a text string
- the pattern state as described in Section 2.3
- the color of the pattern (needed for visualization)

2. Nautilus

Additional to those two files each pattern is saved in a separate file. The pattern file containing the basic pattern informations is loaded and each pattern in it is translated into a DummyPattern. This DummyPattern is used as long as there are no specific data of a pattern is needed. If the algorithm reaches Step 3 for a certain pattern or needs the explanations for visualization the actual pattern is read from its file — identified by the pattern ID — and loaded into the memory discarding the DummyPattern. This way a pattern history is only loaded completely when needed.

Note: Currently a pattern is only identified by its ID that is generated randomly when the pattern is generated. If two patterns are identical but their ID, currently the history does not see them as the same. This needs to be addressed in future if only parts of a history are needed.

I modified the Nautilus to access patterns only by their 64-bit identification number. When Nautilus needs access to one pattern it states a 'request' to a data structure translating the ID to the actual pattern. When the pattern is not yet loaded to memory and the request does only need simple informations about the pattern (i.e. the state, the database view name) then a so called DummyPattern containing only those lower informations is returned. If the request states the actual pattern is needed (e.g. for explanations) then the pattern is read from disc transformed to a pattern-object and returned.

Another idea I worked on is about pattern similarity: If two patterns are similar, can they be computed simultaneously, or can the second one at least reuse the results from the first one?

Example 4: Lets say we a scenario \mathcal{S} containing multiple queries and we get two patterns representing the following two subqueries:

```
select *
  from A
 where a > 5

select *
  from A
 where a > 10
```

As we can see the second query is actually a subset of the first one on the given database table. We could merge the third and the fourth step, the execution on database and solver step, by doing the following:

1. execute the the query of the superset

2. Nautilus

2. check the matching tuples of the subset first

We only need to send one database request instead of two by using the superset and if we check for matching tuples on the subset first we know, that every satisfiable matching tuple for the subset would be satisfiable for the superset as well. We only need to send the not-matching tuples of the subset to the constraint solver to complete the superset. In this example the WHERE-Clause was relatively simple. With more complex queries it is even more complex to reuse data. See another example:

Example 5:

```
select *
  from A
 where a > 5

select *
  from A
 where a > 5
       and a < 10
```

Now we can't use one of both queries to fully answer the other one without additional constraints for the constraint solver. Of course it is possible to use the superset of the first query to answer the second one, but we need to check the $a < 10$ constraint. As [5] shows, the constraint solver is the slowest part of the algorithm and an additional constraint might slow it even more down.

But we could still reuse old data. If we compute the pattern for the second query first we can reuse all informations and explanations and combine them with an adapted query like:

```
select *
  from A
 where a >= 10
```

Now we reduce the amount of work for the constraint solver by reusing the second query and reducing the result set of the first query.

Both scenarios are not only possible with two queries but with n queries although it might get to complex.

The question is not whether it is possible but whether it is reasonable — there might be more overhead. We need the following steps to achieve such reusability:

1. analyze each subquery for compatibility with each other

3. The Visualization

2. find super- and subsets
3. eventually adjust queries to receive missing data from other query
4. find an optimal computation order

For simple scenarios like Example 4 and 5 this might be practicable but most queries have more than two predicates and are far less similar. Actually such problems are *NP*-complete. Another question is: how often do we have such similar queries in a debugging scenario?

Due to time problems caused by a used framework (see 5.1) I was not able to implement and test this feature for simple cases.

3. The Visualization

Basically visualization is used for the representation of informations, data or knowledge. Graphics can present complex information quickly and more clearly than text especially in education and science and are used to communicate concepts. There are several situations where a graphic helps to receive a statement instead of plain text.

Artemis does not visualize the explanations in an advanced way. There is a `TreeList` showing all explanations for a scenario, each explanation split into their tables affected by it together with their tuple. This tuple is either a tuple already in the database or are tuple that needs to be inserted. tables that need new data inserted are marked red and green otherwise. Additional there is a condition entry show all needed conditions for this explanation.

In combination to this `TreeList` one can use a graph to see one specified explanation. Each table and its data tuple the explanation needs is represented as a node of the graph designed as a table. These nodes are connected as the queries of the scenario define incorporating the conditions of the explanation.

To find a certain explanation quickly a list is not always sufficient. Nautilus was extended by two Graphs to give the user an overview of a selected witness and to filter the also extended `TreeList` by their selections within the graphs.

One assignment of this work was to add a visualization. I was inspired by [3] and their fish-eye graphs to represent connections between data in large data sets. I designed a graph using arrows (edges) to show a data connection between tables added the possibility to collapse nodes and edges for readability purposes such as [3] does. I then added a second graph for a better understanding and visualization on a more abstract level. The first graph became the Explanation Set Graph and the second one the Explanation Graph controlling the first one.

3. The Visualization

The Eclipse Plugin implementing the algorithm has now a second perspective, as a group of views is called, containing three Graphs, the List of explanations and a view to choose the witness we want to explore. Figure 2 shows the new perspective.

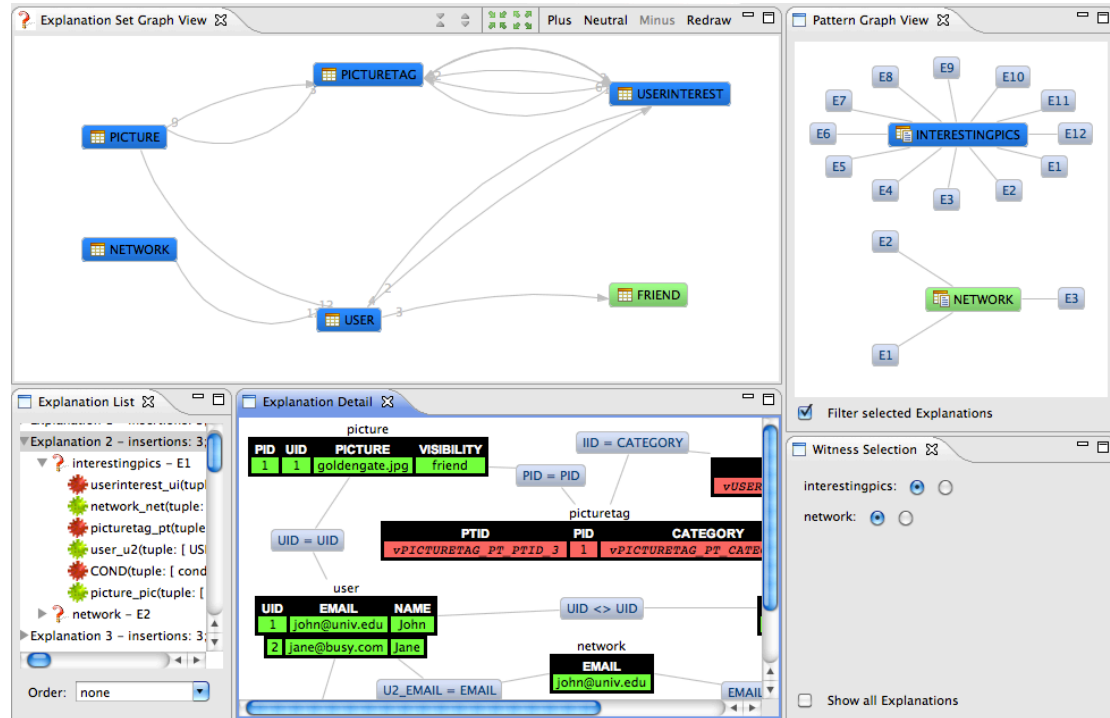


Figure 2: *Nautilus Graph Perspective*: Explanation Set Graph (top left), Explanation Graph (top right), Witness Selection (bottom right), Explanation Detail (bottom middle), Explanation List (bottom left)

3.1. The GUI Elements

The two new graphs, the old Explanation Detail Graph, the Explanation List and a simple Witness Selection view are part of a new perspective for Nautilus (see Figure 2). The idea behind the Explanation Graph (see page 18) is to have a easy overview of sub-explanations and their pattern as well as to control the second graph, the Explanation Set Graph (see page 18). While the Explanation Set Graph has multiple edges for one sub-explanation those edges have exactly one node in the Explanation Graph representing them, making it easier to find them later on.

The five views are hierarchical in their depth of details. The Witness Selection is the most abstract one representing each pattern as a simple radio button grouped by their database view. The Explanation Graph showing the explanations as nodes for each selected pattern and the Explanation Set Graph expanding those explanations as edges

3. The Visualization

connecting tables while indicating an explanation action.

The Explanation List shifts from a graphical visualization to a hierarchical list while displaying for the first time the conditions of the explanations. At last the Explanation Detail View again in form of a graph displaying the explanation tuples and their conditions as an alternative for the List View either in form of a complete explanation or one of its sub-explanations.

In this hierarchy the higher levels always determine the content of the lower ones. Whereas the Witness Selection defines the displayed witness does the Explanation Graph highlight or filter selected (sub-)explanations.

Explanation Graph

The Explanation Graph shows for a selected witness its containing patterns represented as a node. These patterns are identified within the graph by the name of the view they are part of. Remember, each witness contains exactly one pattern for each view in Q . This means there are always the same nodes no matter what witness we select but they can represent a different pattern each time. For each pattern this graph shows all sub-explanations already calculated by the algorithm as a node connected with the appropriate pattern.

Figure 3 shows two patterns with several explanations. Each pattern has a different color, for better association when switching the witness. Nautilus has currently ten predefined colors for patterns.

Explanation Set Graph

The Explanation Set Graph is the second new graph within Nautilus. This graph shows all tables associated with the patterns of a selected witness as nodes. Each sub-explanation for a pattern results in a set of edges that connects the tables of this pattern. These edges are computed via join paths given by the query the pattern is representing (see Example 6)

In theory I planned to give each edge a meaning reflecting the condition of their part within the explanation. If an edge connects two database tuples (and therefore two database tables) with both tuples already within the database instance the edge will be a simple line. If one of the tuples is not yet within the database instance and has to be inserted a pointer should point to the to-be-insert tuple indicating that part of the new tuple results from the other tuple, or if both to-be-inserted are inserted for the same reason.

Originally I intended to give the user the possibility to define a *insertion pattern* controlling the algorithm's explanation computation order, but the used framework made this two difficult if not even impossible (see more in Section 5.1).

3. The Visualization

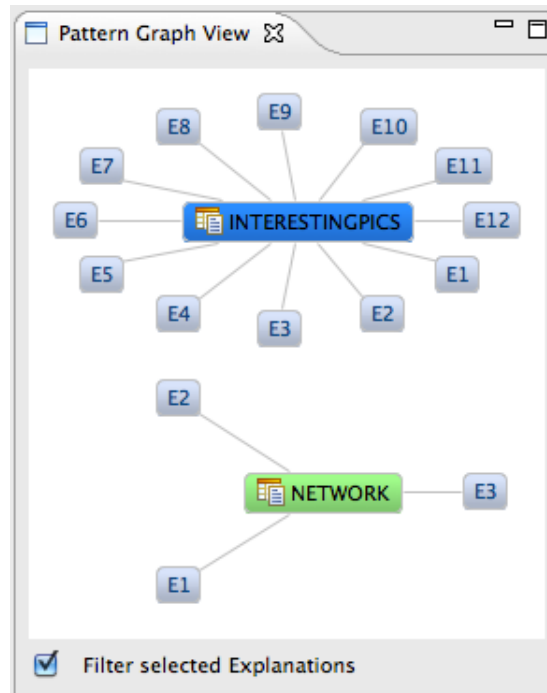


Figure 3: Explanation List

Example 6: A Query like

```
select *  
  from A, B  
 where A.a = B.b
```

would result in an edge between table A and table B because of the predicate $A.a = B.b$.

Each table has the color of the pattern it is referenced by. If a table is used in multiple pattern the first pattern calculated defines the color; see Figure 4. There are the following rules:

1. each join predicate in the query has an edge in the graph
2. an edge is directed iff one of the source and/or target has a tuple to be inserted
3. a directed edge has its pointer towards the table with the insert

Because a graph can get cluttered the more explanations we have for a witness (see Figure 4), there is an option to collapse the graph edge-wise. This option groups all

3. The Visualization

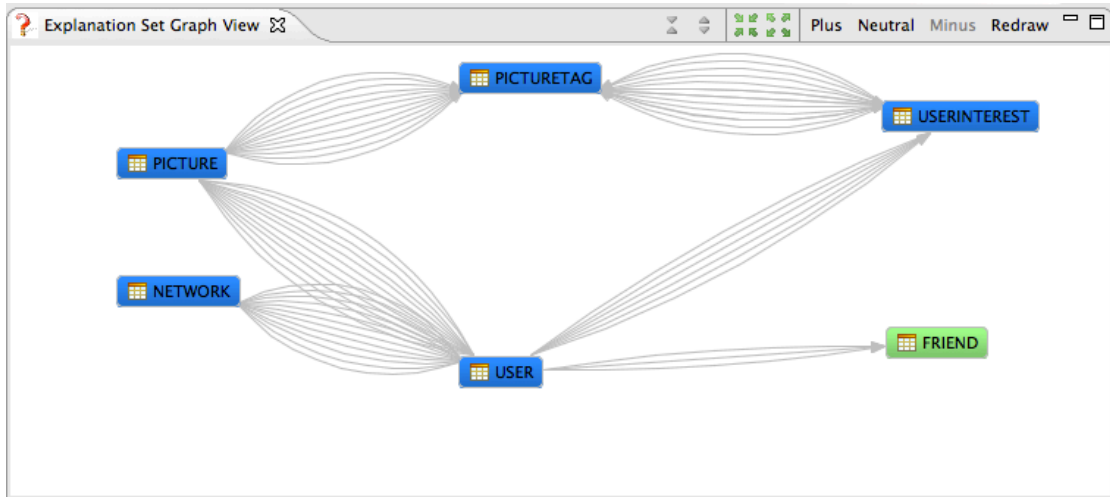


Figure 4: Explanation Set Graph

edges between two nodes and displays only one edge as a representative for each group — *note*: there are at most four groups between each pair of nodes. There is of course an option to expand a collapsed graph back to its original form. Figure 5 shows collapsed edges in the Explanation Set Graph.

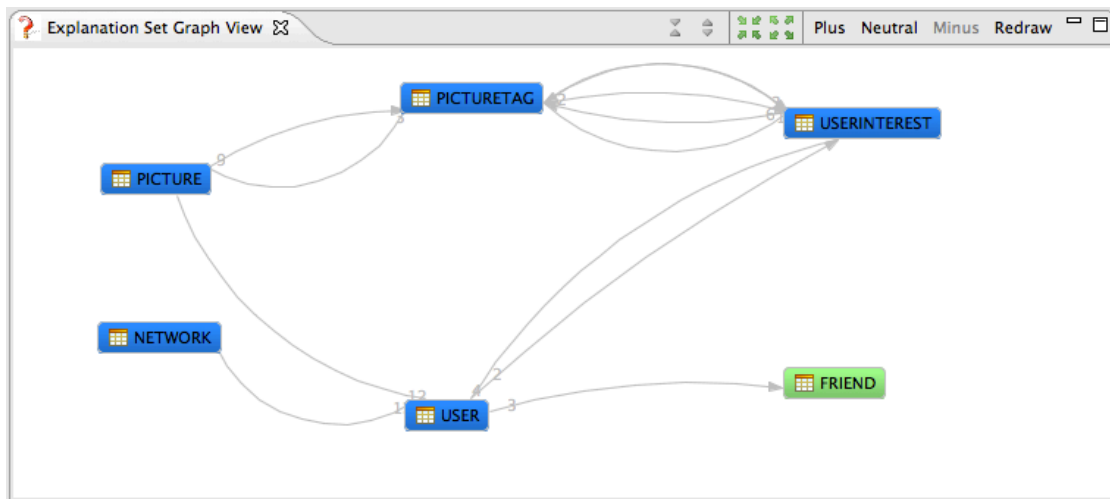


Figure 5: collapsed edges in the Explanation Set Graph

Another way to enhance the read-ability of a graph would be to add the possibility to choose what explanations are shown instead of showing all. There are several ways to add such option. I decided to add this option directly to the Explanation Set Graph. Select the explanations in one of the main graphs then choose to filter them and all non-selected edges are hidden. This filtering is static, meaning any

3. The Visualization

change of the selection the Explanation Graph is not reflected in the Explanation Set Graph. This filtering has no effect to the Explanation List.

An alternate realization could be through the Explanation list (see Page 21) by adding checkboxes for each explanation; selected means edges for this explanation will be visible. Originally I intended to present an option to reduce the amount of explanations to be calculated to the user by letting him choose the upper explanation bound to solve the problem of a cluttered graph. After working with both graphs I decided against it and implemented this selection filtering.

The graph can also be collapsed node-wise. This option removes all non-selected nodes within the graph and replaces them with a place-holder node marked with the asterisk symbol *. Obviously the edges between the collapsed nodes are now invisible, the remaining ones are collapsed. Between the remaining nodes and the *-node are grouped edges representing all outgoing edges to now collapsed nodes. Figure 6 shows the graph from Figure 5 with collapsed nodes.

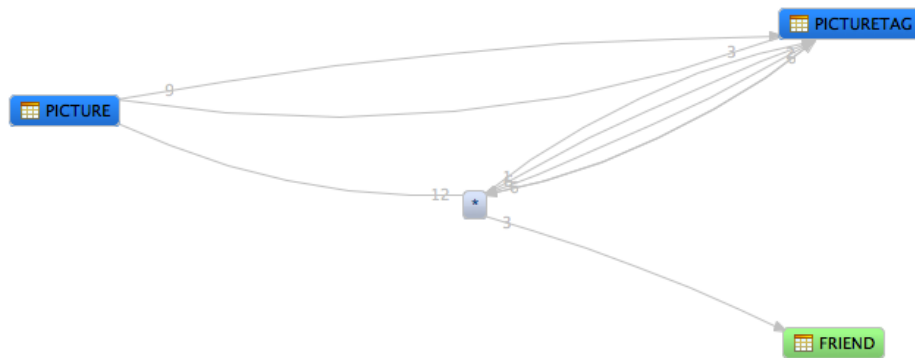


Figure 6: collapsed nodes in the Explanation Set Graph

This option needs at least one selected node to work and less than all.

Explanation List

As mentioned the List with all explanations was extended. This was necessary because of the changes how explanations are calculated. There are two main major changes:

1. only explanations of the selected witness are shown
2. the explanations are unions of their sub-explanations

3. The Visualization

The first change is in according to the two graphs. Both graphs only reflect the selected witness. If the list would show all explanations already calculated, then it could get difficult to find a certain (sub-)explanation in the list for further details.

In Artemis all explanations were computed as one whereas in Nautilus all explanations are unions of sub-explanations. A complete explanation is only build when necessary. For each explanation the list has an entry containing all its sub-explanations. The user is now able not only to inspect a complete explanation but also the sub-explanations. In Artemis, when selecting a explanation in the TreeList a graph — as explained above — showed the details. This is now not only possible for each explanation but also for each sub-explanation.

In addition there is a new way to sort the list on the fly by either the insertion-count of the explanations or the number of side-effects. The sort-order only effects the complete explanations not the sub-explanations.

Because of this sort mechanism the option to sort explanations Artemis presents when starting the scenario calculations are ignored. The option to filter explanations by insertion or side-effects is still functional but is only applied to the Explanation List. This is due to the fact that only this list shows the explanations as one entity whereas the graphs only reflect sub-explanations that can be part of several explanations.

Figure 7 shows the modified Explanation List. We can see three hierarchy levels: the top level of each explanation summarizes the number of insertions and the number of side-effects, the second level shows the containing sub-explanations and their view they are representing and the last level shows the tuples that are to be inserted (marked red) or used (marked green) for this sub-explanation plus the condition (also red).

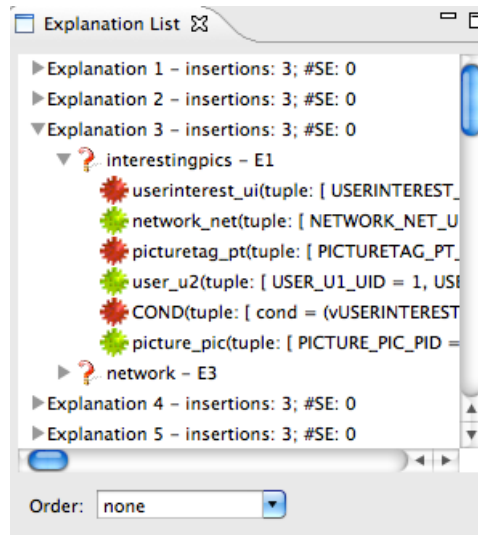


Figure 7: Explanation List View

3. The Visualization

3.1.1. Graph Layouts

Because the layout algorithms part of the used graph framework ZEST (see [2]) did not deliver any usable graph layout I was forced to implement my own.

For the simpler Explanation Graph I was inspired by the fish-eye graph as seen in [3]. In the fish-eye graph nodes are grouped in clusters, each cluster has one center node and all nodes within the same cluster that are connected to the center node are positioned in a circle around it.

The layout algorithm for the Explanation Graph chooses every node representing a pattern as center nodes and all nodes representing an explanation are positioned in a circle around it.

This is a simple straight forward layout that can be computed quickly.

For the Explanation Set Graph I used a similar layout technique. All nodes represent tables used by the patterns so there is no center node. I place all nodes in a circle and count the outgoing edges ignoring multi-edges.

The nodes are placed by the following rules:

1. find a node with minimum outgoing edges — ideally one outgoing edge; ignoring zero
2. place nodes recursively:
 - a) choose one neighbor node
 - b) place nodes
3. if all neighbors are placed and there are still unplaced nodes go to step 1 and place unplaced nodes
4. place nodes with zero outgoing edges last

This way we place neighbors in a cluster, the positioning in a circle guarantees that no edge crosses directly any node making it easier to follow them. This layout does not minimize edge crossing.

This layout is also relatively simple to speed up its calculation. The center position is used for the *-node when the graph is collapsed by nodes.

3. The Visualization

3.2. GUI Interactions

Each of the GUI elements (called a view) interact with one-another. The witness selection view, composed of grouped radio buttons representing the patterns, obviously controls the content the other views display. If the user selects a witness, this view tells the Explanation Graph and the Explanation List what patterns and therefore explanations to use. The Explanation Graph itself controls the content of the Explanation Set Graph.

If a user selects one or more sub-explanation in one of the two new graphs, those sub-explanations are selected in the other graph as well making it easy to see the connection between the graphs and to identify the explanations. Further is there an option in the Explanation Graph (the one with explanations as nodes) to filter all selected sub-explanations in the Explanation List, resulting in a reduced number of explanations. Each remaining explanation contains of at least one of the selected sub-explanations in the graphs. This makes it easier to see more details of certain (sub-)explanations without searching a long list of explanations for the right one.

If the Explanation Set Graph is collapsed edge-wise, then the selection in the Explanation Graph has no influence of the selection in the Explanation Set Graph, but if we select an edge in the collapsed graph, the Explanation Graph highlights all nodes of those explanations that are represented by this selected edge.

If we can reduce the set of explanations in the Explanation List it would be prudent not to allow the opposite. The Witness Selection View allows us to ignore the restriction only to show the explanations of the selected witness within the Explanation List. As a result we will see all (already calculated) explanations of our debugging scenario as in the classic Artemis.

All filters mentioned above as well as the sort-order still works when we see all explanations, but a selected sub-explanation in the graphs will now also be visible in explanations for other witnesses the graphs currently do not show.

3.3. Interactions with the Algorithm

Besides the interactions between the various views there are two ways for the user to interact with the nautilus algorithm.

The first way is through the Witness Selection View. If the algorithm is still running in the background and the user is switching the witness the algorithm is changing the priority of the selected witness, reordering the witnesses for calculation, stopping the current calculation as soon as possible and then resumes calculating with the new witness selected. If this witness is already selected, the algorithm iterates through the witnesses until it finds an unfinished pattern to resume its calculations.

3. The Visualization

Simultaneously the graphs and the Explanation List are rebuild according to the new selection incorporating all already calculated informations.

This way a user can influence the overall calculation order subject to his preferences.

The second interaction is via the Explanation Set Graph. Here a user can select tables and assign a calculation preference such as *'only calculate explanations with an insert on this table'* or *'only calculate explanations with no insert on this table'*. Both assignments influence the algorithm in the solver step. Here the algorithm checks if a matching tuple violates such a condition or if it should pass the possible explanation to the constraint solver.

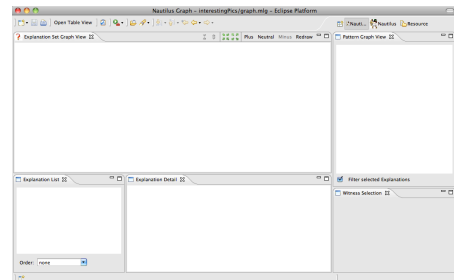
If there is such an assignment the algorithm guarantees at the end to have calculated all explanations even those violating an assignment. When the algorithm gets informed not to calculate all explanations it ignores those in the first run and after calculating all patterns, calculates those missing explanations before terminating.

3.4. Use-Case Scenario

It is easier to understand the concept of these visual elements in an extended example. This use-case is based upon the Interesting Pictures scenario introduced in [5] and [6]. The queries, tables and tuples can be seen in Appendix A.

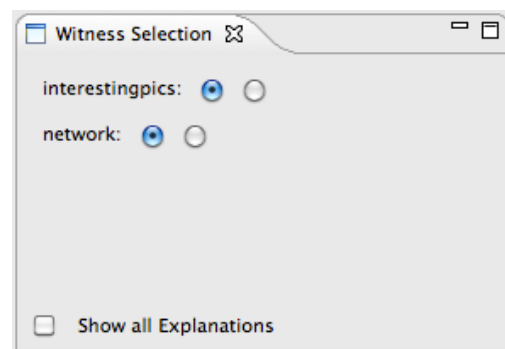
1.

When we start Nautilus the five Views in Eclipse are empty.



2.

After selecting the debugging scenario Nautilus shows the patterns grouped by their query in the Witness Selection View in the bottom right. The patterns of the first witness are automatically selected.

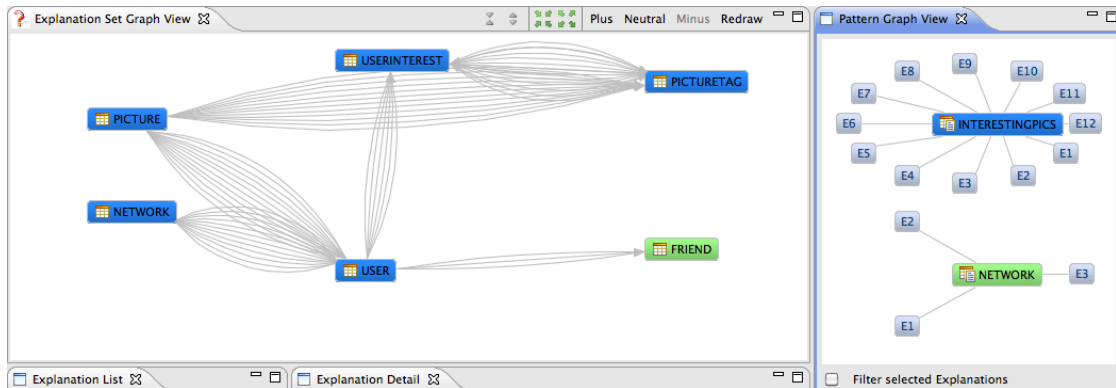


3.

The algorithm starts calculating in the background and both graphs on the top get up-

3. The Visualization

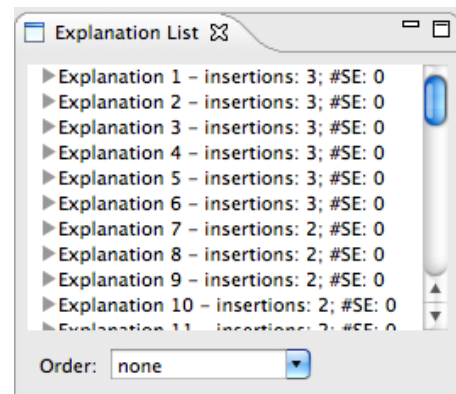
dates. We end with two complete graphs for the selected witness.



4.

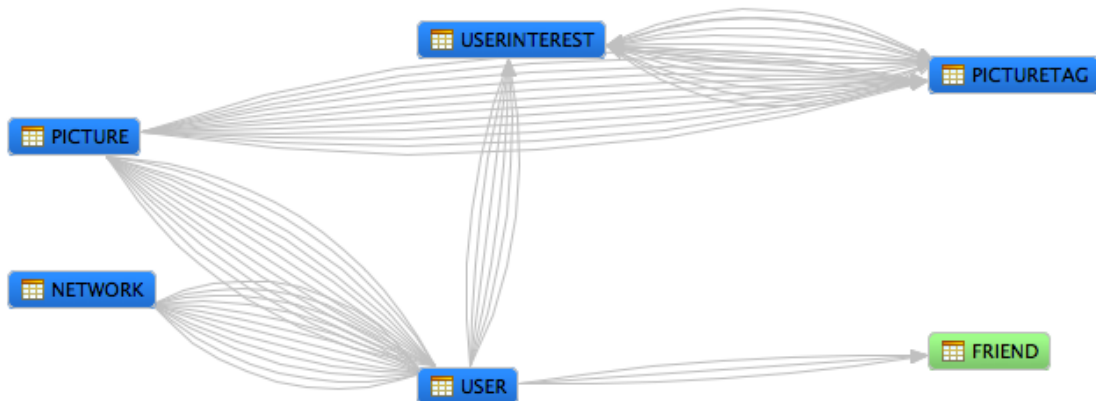
In the same time the Explanation List on the bottom left gets updated.

We see a list of explanation given a number to identify, the number of insertions necessary for this explanation and the number of side-effect.



5.

First we look at the Explanation Set Graph on the top left. We see several nodes representing the tables of our queries used by the current witness. Let's select an edge between two blue colored nodes. I select one edge of the six between *UserInterest* and *User*.

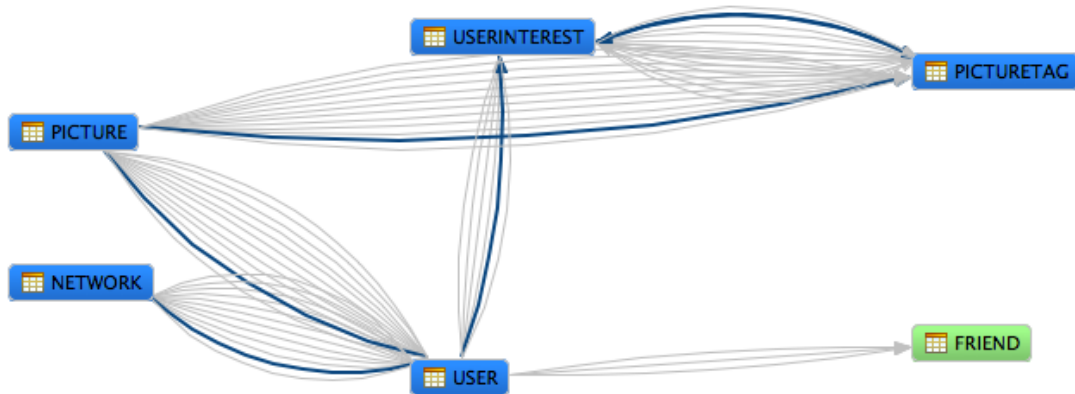


3. The Visualization

6.

Although I just selected one edge five edges are clearly marked selected. Those five edges represent exactly one explanation. We see that two edges have an arrow, both pointing to *UserInterest*. This means in this explanation we need to insert data in *UserInterest* and we have an entry in *User* that is needed in this new tuple.

If we look closely we see two more arrow-points pointing to *PictureTag* but other edges are overlapping.

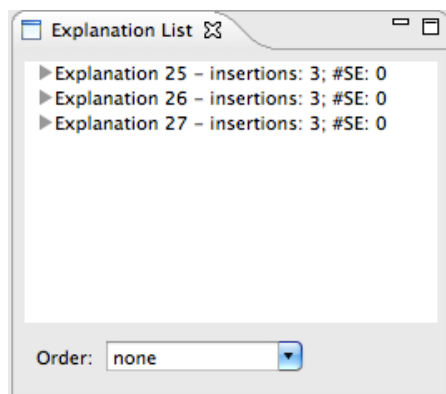
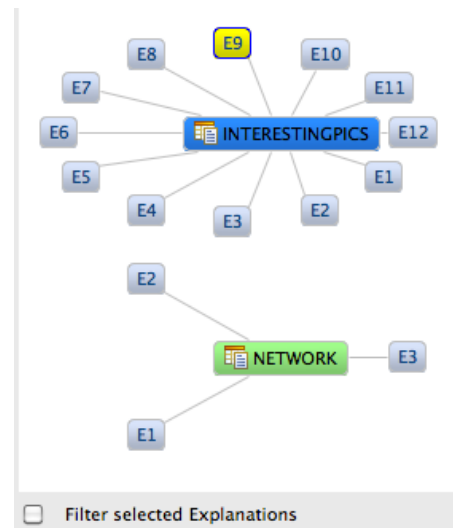


7.

The selection I made is reflected in the Explanation Graph on the top right. The explanation E9 of pattern *InterestingPics* is highlighted yellow.

Lets select the *Filter selected Explanations* option underneath the graph.

This option will filter the Explanation List only to show explanations containing E9.



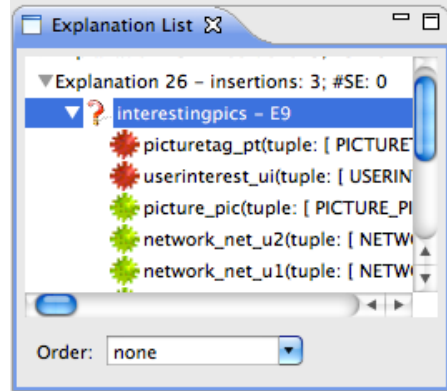
8.

The Explanation List is reduced to three explanations. In the Explanation Set Graph we saw three edges from *User* to *Friend*. *Friend* is green same as the *Network*-Pattern. The Pattern has three sub-explanations. Those three sub-explanations each are paired with E9 currently selected, therefore we see three explanations as a result of the filter.

3. The Visualization

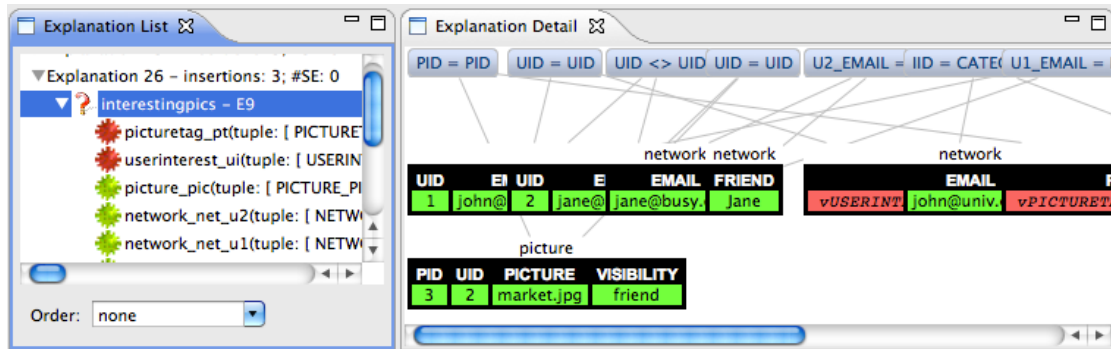
9.

We take a look deeper into the Explanation List and open Explanation 26. We see a hierarchy level named *InterestingPics - E9*. So we see this explanation really contains E9. The next hierarchy level shows us *PictureTag* and *UserInterest* marked red, meaning we need to add tuples to both tables, but we already know this from the Explanation Set Graph.



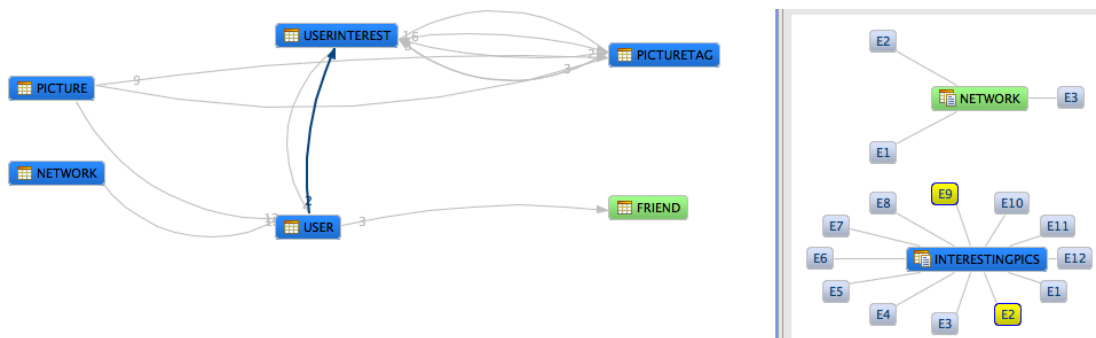
10.

When selecting E9 in the Explanation List the view on the right shows the sub-explanation as graph. This graph is different than the Explanation Set Graph. Here we see the tuples of each table visualized as a node connected by conditions. This graph is unchanged from Artemis.



11.

Back to the Explanation Set Graph, we want to see if there are more explanations like E9. So we collapse the graph and then select the arrow pointing from *User* to *UserInterest*. We see in the Explanation Graph, that there is a second sub-explanation with this type of insertion — sub-explanation E2.



4. Tests and Evaluation

The visualization helps us to analyze the explanations, we can find similar explanations and we can choose the abstraction level we want for a sub-explanation. We can reduce the amount of explanations we see by filtering and we can influence what certain views show by selection.

4. Tests and Evaluation

4.1. The Debugging Scenarios and Test Platform

The scenarios Pic1, Pic2, Pic3 and Pic1,2 are based upon the Interesting Pictures from [5] and [6], whereas the tpch scenarios use two queries from the TPC-H query set. The tables and queries for the Interesting Pictures scenarios can be seen in Appendix A.

Pic1 uses the Network view with the missing tuple

$$\langle \$email1, \$name, \$email2 \rangle$$

and Pic2 uses the UserNameInterest view with

$$\langle \$name, \$cat \rangle$$

Pic3 utilizes two views for a combined scenario the Network view and the InterestingPics view. As you can see in Appendix A the InterestingPics view uses the Network view as a source table. For this debugging scenario the tuples

$$\text{Network} \langle \text{john@univ.edu}, \$friend, \$email \rangle$$

and

$$\text{InterestingPics} \langle \text{john@univ.edu}, \$picture, \$friend \rangle$$

. We don't want inserts in source tables User and Picture, so we have $Q_{im} = \{\text{User}, \text{Picture}\}$
Pic1,2 is a combination of the queries from Pic1 and Pic2 with the tuples

$$\text{Network} \langle \$email, \text{John}, \text{john@univ.edu} \rangle$$

and

$$\text{UserNameInterest} \langle \text{John}, \$category \rangle$$

. Source table User is set to *fixed* and therefore being part of Q_{im} as in Pic3.

I ran the tests on a MacBook containing an Intel Core 2 Duo 2.26 GHz with 4 GB DDR3 memory. The database is a local installation of IBM's DB2 Express-C 9.5.2 for Mac. Each test case was executed several times with Artemis and Nautilus, all results are average values. Each Nautilus result was compared to the Artemis results.

4.2. Tests and Evaluation

The first test I ran was to see whether or not the changes made to the algorithm structure, mainly the pattern oriented architecture, has any impact on the speed of the four steps. For this test I disabled the new GUI elements. course, I did not expect to see any speed improvement in step 1. Table 1 shows the runtimes for each major step of Nautilus in comparison with Artemis. As we can see Step 1-3 do not gain a speedup measurable, but Step 4 has some interesting results. Pic1 is about 50% faster in Nautilus, whereas Pic2 and tpch query 7 don't show any noticeable speedup. Pic3 scenario shows exactly what I wanted to see. As explained above Pic1 and Pic2 are composed of one query and one tuple, but Pic3 has two queries and two depending tuples. This would allow the new design to reuse past calculations because a pattern is used for several witnesses and as we can see Nautilus is in this case roughly 4.4 times faster than Artemis.

	Pic1		Pic2	
	Nautilus	Artemis	Nautilus	Artemis
Step 1	0.01	0.00	0.01	0.00
Step 2	0.02	0.04	0.05	0.01
Step 3	0.02	0.03	0.01	0.00
Step 4	13.21	31.13	3.26	3.05

	Pic3		Pic1,2	
	Nautilus	Artemis	Nautilus	Artemis
Step 1	0.00	0.00	0.00	0.00
Step 2	0.07	0.03	0.05	0.01
Step 3	0.01	0.01	0.01	0.00
Step 4	3.62	16.06	2.15	6.53

	tpch7		tpch19	
	Nautilus	Artemis	Nautilus	Artemis
Step 1	0.01	0.00	0.02	0.01
Step 2	1.60	1.50	0.07	0.09
Step 3	70.61	69.96	0.02	0.02
Step 4	4992.43	5098.34	5.18	1.8

Table 1: Runtime in seconds for each step for several debugging scenarios

If we take a closer look at tpch query 19 we see that this query is the union of 24 subqueries. This high number of subqueries reduces the speed of the pattern oriented way in this case by 280% this is because the higher calculation cost of one pattern.

This test case shows clearly that, the advantages are only visible when using multiple queries and tuples, but we do not see a disadvantage as long as the query isn't composed of a extreme high number of subqueries.

4. Tests and Evaluation

We also see that step 4, the solver step, is responsible for more than 90 per percent of the runtime. Every speedup in step 1-3 will not be noticeable compared to step 4.

Next I was interested in the overall runtime of both algorithm variations. For this I compared Nautilus without GUI, with GUI and Artemis. Table 2 shows the results.

	Nautilus		Artemis
	no GUI	GUI	
Pic 1	13.31	116.81	31.24
Pic 2	3.37	40.79	3.07
Pic 3	3.76	31.10	16.14
Pic 1,2	2.92	19.41	6.56
TCPH 7	5173	—	5280
TCPH 19	244.8	—	31.68

Table 2: overall runtime comparison in seconds

Overall we see the same result as in Table 1. Nautilus (no GUI) has the same runtime as Artemis when dealing with one database view containing few subqueries.

A bit frustrating is the runtime of Nautilus with visible GUI. As we can see in Table 2 Nautilus with GUI is nearly ten times slower than Nautilus without GUI. Those values are the result of multiple threads on a two core machine combined with suboptimal scheduling. Each step itself does not take more time, but the calculations of the graph interrupts the calculations of the algorithm. Of course the algorithm, the GUI and the graphs have their own threads but threads can only work parallel if the underlying machine has enough resources. See more in Section 5.

Now, let's see how long it takes for the first explanations to be calculated. As noted earlier I expected to see a minimal slowdown for the first explanation but a drastic speedup for the later ones. We see the speedup in scenario Pic1 and Pic3.

Figure 4.2 shows a diagram containing the time it takes till the first explanation is ready and the time the scenario takes to be done completely, comparing Nautilus (no GUI) and Artemis (for better visualization the diagram cuts at 4 seconds calculation time).

Pic1 and Pic2 — having one query each — show us that the first explanation takes a few milliseconds longer in Nautilus than in Artemis; I can only guess this is due to more calculations done beforehand i.e. search for history, create advanced data structures.

Pic3 though faster in the long run is alarmingly slower in computing the first explanation. The explanation is simple and there are a few possible solutions to reduce this latency I will address in Section 5.

Pic1,2 confirms the results from Pic3, again it takes longer for the first explanation but is noticeably faster to compute all explanations.

5. Disadvantages, Problems and possible Solutions

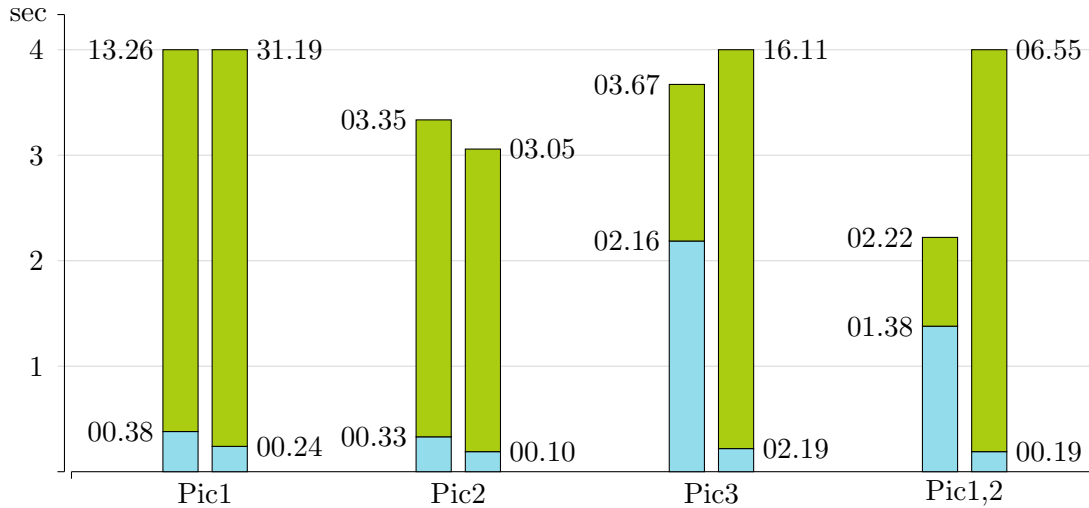


Figure 8: runtime comparison: time to first explanation and until algorithm terminates — first column representing Nautilus (no GUI); second column Artemis

5. Disadvantages, Problems and possible Solutions

While in theory my changes sound good, there are some disadvantages and problems when implementing. The first disadvantage of those Nautilus changes over Artemis is the slowdown of the first explanation, meaning, the pattern oriented approach needs more time before the first explanation is finished calculating if the constraint solver is slow as seen in Section 4.2 and Figure 4.2. A solution is to either speed-up the constraint solver or, if not possible, *not* to calculate each pattern completely before switching to the next one but to calculate only a *small amount* of explanations for one pattern before switching to the next pattern within one witness in a round robin procedure until the witness is completely calculated. This could not only be achieved with an *iteration model* but with an *parallel-programming model* (and a qualified machine) as well.

Another problem shows on slower machines with few cpu cores like mine. The time needed to generate the various graphs and the interaction of the GUI elements can take a lot of cpu time and therefore reducing the free cpu-time for the algorithm resulting in a longer runtime overall (see Table 2). The only way would be to take full advantage over multiple processors by assigning the GUI threads and the algorithm threads to different processors. Otherwise one or both threads will always suffer. Modern CPUs like Intel's Core-i7 might not have such noticeable extreme slowdowns due to having four core and hyper-threading technology, meaning each core can run two threads of the same process, making it an overall of eight simultaneously threads. I had no chance to test Nautilus on such CPU.

5. Disadvantages, Problems and possible Solutions

Another speed related problem occurs on the GUI: to do all necessary calculations for displaying the graphs so a user can inspect the explanations and interact with the algorithm takes time, so the user will not always have the feeling to see the explanations as they come. Instead the results will come quite some time after they are calculated when new results are ready to be displayed. This can result — again on slower machines or when badly scheduled — in a blockage of the old visual result by the new one meaning the user has to wait until everything is calculated because the new graph invalidate the old one.

Example 7: Each view has its own thread; two graphs calculated simultaneously together with the Explanation List plus the algorithm in the background, that makes four Threads - assuming the graphs itself don't have their own threads and the database accessed by the algorithm does not need any cpu time. On a two-core processor like mine, they are 'battling' non stop for much need cpu time.

This happens especially when multiple threads are running in combination with complex debugging scenarios. On smaller scenarios I was able to see the effect of getting the visual additions like new nodes and edges 'as they come' even on my machine, thats why I assume its again just a speed-related problem.

Therefore the solution would be again: more cores, optimized scheduling.

5.1. Graph Implementation

To implement graphs Artemis used a framework called Zest (see [2]). Zest — The Eclipse Visualization Toolkit — is modeled after JFace to be as close to the Eclipse Plugin Standard as possible and comes with the basic Graph Utilities and various layout algorithms. During implementing the two new graphs I had many difficulties with this framework. A lot of methods I needed to adjust to achieve various features where declared as private and therefore protected from being overwritten, resulting in suboptimal solutions.

The biggest problem I had was finding a working layout algorithm. It seems the layout algorithm are optimized to be used with so called Label- and ContentProviders. The Content Provider is building a graph from various object I pass as parameters. Normally this means I define several objects as a base for nodes and a data structure containing the informations about the connectivity of those objects within the graph. By doing so I loose the control over the resulting graph nodes and edges. For Nautilus I need the absolute control over all nodes and all edges resulting in abandoning the Providers when necessary and breaking a lot of Zests features.

Before a layout algorithm is applied to a graph, the connections between the graphs where analyzed to be included in the layout calculations but not when using the Providers; all I got where empty information objects. I had to re-implement those analyzing steps when needed. This took me time to implement and slows the layout algorithm down

6. Conclusion

being redundant work.

Another problem with the layout algorithm seems to be that the calculations are just positioning the node ignoring the edges. As a result the edge crossing is far from optimal.

Another problem dealing with Zest was having no influence on an edge when selecting such in case of redrawing. When an edge gets selected it gets highlighted in another color, but when dealing with a lot of edges between a pair of nodes it is probable that the selected edge is partially covered by other non selected edges and therefore not recognizable enough.

For example when dealing with directed edges, the actual pointer might be hidden underneath a dozen other edges and as a result not visible. I was not able to tell Zest to redraw any selected edge on top of all non-selected edges, in fact I had no control over drawing the edge layout and drawing algorithm at all besides the possibility to define a bend and the color.

When dealing with graphs, an edge has a certain meaning resulting in a certain look, like having an pointer, an diamond or nothing at the end (and/or beginning) and maybe a label with a name or the weight of the edge. The pointer or diamond can be hollow or black probably having different meanings like in UML. Zest supports either a simple line (connected, dotted or dashed) or a directed line having an simple pointer. But I originally intended to have not only one pointer, but also a hollow one and bi-directional edge. The later one I realized with two edges overlapping each other pointing in the opposite direction. The hollow pointer got lost during implementation. I also had no control over the label, basically supported but with no way to adjust its position.

A lot of feature sounding good in theory are not practicable with Zest as a graph framework. I suggest switching to swing having more frameworks or using a non-open source graph framework for SWT.

6. Conclusion

There are four kinds of optimizations I have implemented. First the structural changes to reduce the calculations by splitting witnesses to introduce a reusability of a pattern for multiple witnesses. The second is the calculation order like the order of the witnesses to reduce time. Third, we have the the possibility of prioritizing like changing the order by prioritizing another witness then the one in the queue. Fourth we have the restructuring to a iterative model instead of an *all-or-nothing* model.

Those four optimization types are combined to achieve the various features I presented in this diploma thesis, for example the capability to save and load debugging scenarios utilizes the iterative model and takes advantage of the structural changes made for pattern reusability.

6. Conclusion

The test cases show that those optimizations reduce in fact the runtime of Artemis. The question is, whether those optimizations not only have an effect on the fourth step of the algorithm but also in the first three steps.

Sadly the runtime effect of the optimizations is spoiled at least on slower machines by the visualization of the explanations.

Having an graphical interface and the ability to interact with the algorithm has the possibility to maximize utilization of complex data, but without proper hardware the tradeoff would be the runtime.

A. Database

The following are informations about the used databases and queries used for the debugging sceanrios.

A.1. Photoshare

Photoshare is an example scenario introduced in [6] and used for several test sceanrios in [5].

Source Tables

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

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

PictureTag		
<u>PTID</u>	<u>PID</u>	Category
PT1	P1	I3
PT2	P1	I1
PT3	P2	I4
PT4	P4	

UserInterest

A. Database

<u>IID</u>	<u>UID</u>
I1	U1
I4	U1
I2	U2
I3	U3
I4	U3

Friend	
<u>UID1</u>	<u>UID2</u>
U1	U2
U2	U3

Queries

Network-View:

```
select u1.email, u2.name, u2.email
  from user as u1, user as u2, friend as f
  where f.uid1 = u1.uid and f.uid2 = u2.uid
union
select u1.email, u2.name, u2.email
  from user as u1, user as u2, friend as f
  where f.uid2 = u1.uid and f.uid1 = u2.uid
```

Result of the Query:

User		
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

InterestingPics-View:

```
select u1.email, pic.picture, u2.name
  from user as u1, picture as pic, userinterest as ui,
  picturetag as pt, user as u2
  where ui.iid = pt.category and pic.pid = pt.pid
  and u1.uid = ui.uid and u1.uid <> u2.uid
```

A. Database

```
and u2.uid = pic.uid and pic.visibility = 'public'
union
select net.u1_email, pic.picture, u2.name
from network as net, user as u1, picture as pic,
userinterest as ui, picturetag as pt, user as u2
where net.u1_email = u1.email and ui.uid = u1.uid
and net.u2_email = u2.email and u2.uid = pic.uid
and pt.pid = pic.pid and pt.category = ui.iid
and pic.visibility = 'friend'
```

Result of the Query:

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

References

- [1] *Eclipse*. <http://www.eclipse.org/>
- [2] *Zest: The Eclipse Visualization Toolkit*. <http://www.eclipse.org/gef/zest/>
- [3] *Visualizing large-scale RDF data using Subsets, Summaries, and Sampling in Oracle*. 2010 . – 1048–1059 S.
- [4] CHAPMAN, Adriane ; JAGADISH, H. V.: Why not? In: *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*. New York, NY, USA : ACM, 2009. – ISBN 978-1-60558-551-2, S. 523–534
- [5] HERSCHEL, Melanie ; HERNÁNDEZ, Mauricio A.: Explaining Missing Answers to SPJUA Queries. In: *Proc. VLDB Endow.* (2010)
- [6] HERSCHEL, Melanie ; HERNÁNDEZ, Mauricio A. ; TAN, Wang-Chiew: Artemis: a system for analyzing missing answers. In: *Proc. VLDB Endow.* 2 (2009), Nr. 2, S. 1550–1553. – ISSN 2150-8097