Master’s Thesis Nr. 31

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

Credit Suisse

User Feedback Integration - Incremental Improvement

by

Mike Klausmann

Supervised by

Prof. Dr. Donald Kossmann and Lukas Blunschi

March 2011 – September 2011
Abstract

Keyword search systems became very popular in the last decade because they allow people to access and find information in an easy and comfortable way. However, to keep the quality of these systems on a high level, the system designers have to address several problems like changing requirements of the users or changes in the underlying data structure. We believe that user feedback is an elegant way to handle some of these problems and keep the system flexible and adaptable for future changes in the requirements. To prove this concept this report explains the design and implementation of several user feedback features which are added to an existing keyword search system over data warehouses. We will show by example how these features can be used by the users and how they improve the usability and flexibility of the given system. At the end of this report we will show proposals for other feedback features and how the implemented features could be further improved.
1 Introduction

1.1 Motivation

Keyword based search systems are very popular nowadays because they offer an easy way, even for non expert users, to access and browse information which is part of a huge and complex data set. These systems are commonly used for Internet search, but also a lot of research exist to provide the same functionality on other information pools for example large relational databases like data warehouses ([2], [9], [8], [5]) or semi structured data like trees or other graphs ([7]). However, because of changing user requirements or changes in the underlying data structure these systems have to be very flexible and strongly maintained to maintain the quality of their service. We believe that user feedback provides a good way to solve these requirements in an elegant way because it gives the system users a direct way to adapt and improve the system which they use. The goal of this thesis is to show different kinds of user feedback features and demonstrate how they can be used to improve the usability and quality of such a keyword based search system.

1.2 Keyword Search System Description

1.2.1 Design

The keyword search system which we use in this thesis works on relational data warehouses (DWH). It takes a keyword search query as input and returns a list of all SQL statements which match the given query. To bridge the gap between the “natural language” of the keyword query and the “technical language” used in the data warehouse design the system uses a metadata graph. The metadata graph is a connected, directed and acyclic tree where leaf nodes have a link to the corresponding relational base data (See Figure 1). The graph consists of several subcomponents which are described in the following.

**Domain Ontologies:** Domain ontologies containing expressions which are commonly used in the company for queries to that DWH. The domain ontologies are used to match these expressions to expressions used in the Conceptual Schema of the DWH. As the name implies, domain ontologies are specific to the domain in which they are used. Because of that, domain ontologies can be reused for systems with the same domain.

**DBpedia:** A service offered by wikipedia used to identify commonly used expressions and synonyms and match them to expressions used in the Conceptual and Logical schema. In contrast to the domain ontologies the same DBpedia data source is used for all DWH keyword search systems.

**Conceptual Schema:** High level view on the DWH often used by business personal to express which data they are interested in. The schema is technically not very complex it is therefore usable for people with no technical background.

**Logical Schema:** The view of the DWH used by the constructors and administrators of the DWH during development and maintenance. This schema is far more complex than the conceptual schema and people need a technical background to understand it.
**Physical Schema:** The schema as it is in the “real physical world”, includes optimizations which increase the performance of the DWH on the hard disk. Because of this complex performance optimizations only technical experts understand this schema. This schema is specific for the database management system in use.

The mapping of the syntax of a keyword query to a the technical syntax of the underlying DWH system is called “Query Classification”. A word or a small sequence of words in the keyword query can be interpreted as:

- **Business Object**: Identified by searching the metadata graph. Later used as entry point to the relations which are used in the specific query.
- **Operators**: All words and word combinations are checked against a pre-defined list of words to identify operators.
- **Values**: Values are identified, by searching the base data for matching entries. A keyword term can be both a business object and a value.
- **Unknown**: All words which do not match to one of the above cases are classified as unknown and are ignored further on.

The system uses an algorithm which has several steps. Each step takes the output of the previous step and the metadata graph as input. The output of each step is a query graph which is refined and enhanced in every step (See Figure 2).
**Lookup phase:** The keyword query is analyzed to identify the business objects, operators, values and unknowns and find the entry points to the Meta Data Graph with them.

**Rank and Top N:** Given the result of the Lookup phase, a first scoring is done which is added to each solution. All solutions are ordered by their score and if there are to many solutions only the top n are handed over to the next phase.

**Tables and Joins:** This phase uses the metadata graph and the discovered entry points to identify the tables which are included in each solution. Furthermore relations between the found tables are detected to create the join conditions for each solution.

**Filters:** Filter conditions are identified by traversing all paths of the metadata graph, from the entry point to the logical attribute and collect all filter criteria which are attached to the nodes on the way. The filter conditions are added to the solutions.

**SQL:** The given query graph is taken and for each solution one or more SQL statements are generated. The output of this phase is a list of SQL statements together with a score.

**Grouping:** The generated SQL statements are compared and grouped to remove duplicate results the scores are combined by computing the average.

---

**Figure 2: Algorithm Flowchart**
1.2.2 Problems

The following section describes a selection of issues, that could be improved in the system described above:

a) The used scoring function in the rank and top top n phase is very crude so it is quite possible that a wrong result gets a high ranking and a good result gets a low ranking.

b) The users of the system (which actually decide what results are good) have no influence on the score of a result.

c) There is no possibility to enhance or adapt the metadata graph at runtime.

d) Users have to use the keywords which are recognized by the metadata graph, there is no possibility for them to add synonyms or add new words.

e) The generated SQL statements always show all columns of the joined tables and the returned columns and rows can not be ordered. So the returned results are sometimes hard to understand, even if they are correct.

1.3 Project Description

The goal of this thesis is to show how user feedback can be used to increase the usability and flexibility of a keyword based search system. We describe the implementation and integration of several user feedback features in the system described in section 1.2.1 and show how these features solve the problems described in section 1.2.2.

1.4 Contributions

The features we add to the keyword search system give the system users the possibility to:

- Rate a result to influence its score and the score of similar results. To be more precise we enhance the result interface with the possibility to rate a result as correct, so so and wrong. These votes are stored and used to compute a user feedback score for each result. The computed user feedback score is used, in combination with the other scores, to compute the overall score of a result.

- Rate single relations to influence the score of all results that contain a given relation. Again we enhance the result interface, with the possibility to rate single relations. The votes are stored and used to compute a relational user feedback score for each result. The relational scores again flow in the overall score of the results.

- Make the system recognize new keywords by adding new nodes and edges to the metadata graph e.g. let the user create new entry points to the metadata by extending the metadata tree with new nodes. The user can create new nodes from keywords in his query which the system classifies as “Unknown”. To connect these new entry points with the graph, the possibility to add outgoing edges from new nodes is integrated in the search system.
• Propose edges and nodes which can be removed from the metadata graph. The system stores these proposals and presents them to the admin of the search system. The admin decides if he/she wants to perform the proposed changes on the metadata or not.

• Set/Change the projection and order by criteria for relations e.g. give the users the possibility to select a relation from a result and change its projection and order by criteria. The selections taken for one relation, also influence the projection and order by criteria of other “similar” relations, for all users of the system.

For every feature we will show how it is implemented and integrated into the given system and give examples to show how they improve the quality and flexibility of it.
2 Like Feedback

2.1 Overview

The like feedback features, which we added to the search system give the users the possibility to rate either complete results or single relations of results. The so gathered ratings flow into the computation of the result scores. The search system uses these scores to filter results, if to many possible results exist, and to order the results when they are presented to the user. The two features of like feedback (complete result and relation specific feedback), are described in sections 2.2 and 2.3. Examples how these features are used are given in section 2.1.1.

2.1.1 Examples

In the following section we will give an example of how the like feedback works in the search system and how it is used.

Let's assume the user has sent the query “address of sara” to the system. In our test environment the search system finds two possible result statements (see Figure 3).
Figure 3: The result page of the search system after the query “address of sara”.

Every result in the result set has three buttons I Like, So so and Wrong. If the user presses the I Like button of result [0], the result gets a like vote in the entry point specific feedback. Because a like vote indicates that all relations in the result are correct, also a positive relational vote for all relations is given. As you can see in Figure 4 the vote has influenced the score of both results. Result [0], which had a score of 0.56 has know a score of 0.85. Result [1, 2] had a score 0.59, which has changed to 0.69 after the like vote. The vote influences the score of both results, because result [0] also contains one relation contained in result [1, 2].
Figure 4: The result page of the search after I Like vote for result [0].

Also the order of the results has changed in the presentation, because the score of result [0] is now higher than the score of result [1, 2]. If now the So so button of result [1, 2] is pressed, the user tells the system that this solution is not completely wrong but contains or misses parts which do not/do belong to the optimal solution. So a so so vote is given in the entry point specific feedback, but no vote is given for the relational feedback. The reason for this is that the system can not decide which relations of the result are good and which are bad. Because no relational vote is given, a so so vote only influences the score of the voted result in a result set. As you can see in Figure 5 the score of result [1, 2] changes from 0.69 to 0.545.
Figure 5: The result page of the search after So so vote for result [1, 2].

If the user hits the Wrong button on result [1, 2], a wrong vote is given for the result on the entry point specific feedback system and all relations contained in the voted result get a negative relational vote. As shown in Figure 6 this again affects the score of both results because they both share one relation. The score of result [0] changes from 0.85 to 0.67 and the score from result [1, 2] changes from 0.545 to 0.335.
Figure 6: The result page of the search after Wrong vote for result [1, 2].

Figure 3 also shows that every relation in the results has two buttons (+ and -). This allows the users to give votes for individual relations. Figure 7 shows the results after a positive relational vote for the relation contained in both results has been given. Again the scores of both results are affected the score of result [0] changes from 0.67 to 0.73 and the score from result [1, 2] changes from 0.335 to 0.375.
2.2 Entry Point Specific Feedback

2.2.1 Description

The entry point specific feedback, basically holds a persistent table where each row stores the votes for one result (See Figure 8). The feedback is called entry point specific, because it uses the identifiers of the entry points as key to identify results. A user can choose between 3 types of votes for a result:

- **Like**: If the result is exactly what the user has searched for.
- **So so**: If the result is not completely wrong, but also not matches all expectations of the user.
- **Wrong**: If the result is completely wrong.
We designed this kind of feedback in a way, that it is accessible in an early stage of the search systems result finding process. Because of that, it can be used to filter results if too many possible results have been found. This feedback feature also recognizes the dependencies of results, which have been generated by similar queries (e.g. “address of Sara”, “address of Mike” or “Adresse von Sara”).

### 2.2.2 Key Description

In the following section it will be explained, how the key for the entry point specific feedback is constructed.

The key of a result consists of the concatenated identifiers (source links) of the entry points that lead to this result (see Figures 9 and 10).

![Figure 9: Format of an entry point specific feedback key.](image1)

```latex
<source link ep1> <source link ep2> ...
```

![Figure 10: Example of a source link.](image2)

http://dbpedia.org/resource/Adressen

One entry point can either be a business object (if it is a node in DBpedia, the domain ontology or one of the schemas e.g. the name of a column), or it can be a value (if it matches an entry of the relational data e.g. the name of a person). To make the key recognize the similarity of results in different result sets, we generalize the source links we use for the key. Only business object source links, which are located in one of the three schemas of the metadata graph are used for the key. Entry points which are not located in the schema, for example ontology nodes, are transformed, by following their outgoing edges until they all reach nodes, which are located in one of the schemas. Then, the source links of the found schema nodes are used for the key. This has the advantage, that the key also recognizes the equality of two results, if the keyword query, which was used to find them, used synonym keywords. Also the source links of the values have to be generalized, to allow the system to recognize the similarity of queries, which just use different values e.g. “address of Sara” and “address of Mike”. The search system that we use, automatically creates a node for each
value, which was queried before, and connects it with the node representing the column where the value is located in the relational data (see Figure 11). So to solve our problem we simply use the source link of the column node instead of the value node source link (see Figure 12).

![Diagram of data structure](image)

Figure 11: Meta data for query address of Sara Guttinger

http://example-world.ch/basedata/individuals/firstname/Sara

http://example-world.ch/basedata/individuals/firstname/

Figure 12: Example of a not generalized and a generalized value source link

This generalization technique enables the feedback system to recognize the similarity of two query results without complicated mathematical procedures. Another advantage of the entry points is that they are accessible in an early stage of the result finding (namely right after the lookup phase see Figure 2). Because of that, the system has access to this feedback also in the first stages and can use it to filter results, if the found result set is to big to process all results.

2.3 Relation Specific Feedback

2.3.1 Description

The relation specific feedback like the entry point specific feedback holds a persistent table to represent the feedback given by the users (see Figure 13).
The table consists of a key column, which identifies the rated relation, and two columns which store the numbers of positive and negative votes. The so gathered ratings influence the scores of the results containing these relations. This has the advantage that, if the user rates one relation in a result, also the score of all other results which contain the relation is influenced. Because of that the user can pick the relations which according to his/her opinion should be part of the result and give them a positive vote. The feedback system now increases the scores of all results which contain these relations so they appear earlier in the result presentation. The system also takes into account how many none rated tables are in a result, so a result which contains exactly the positive rated relations will get a higher score then one which additionally contains several not voted relations.

<table>
<thead>
<tr>
<th>Key</th>
<th># Like Votes</th>
<th># Wrong Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 13: The relation specific feedback table.

2.3.2 Key Description

The key we use for the relation specific feedback, is a text encoding of the join set of the relation. If for example a relation is generated by joining the tables “Address” and “Individuals”, the corresponding relational key would look like this

<Address, Individuals>

This key has the advantage, that it recognizes tables according to their structure and not their content. To give an example, the tables shown in Figures 14 and 15 are results of the queries “address of sara” and “address of mike”.

<table>
<thead>
<tr>
<th>FIRSTNAME</th>
<th>LASTNAME</th>
<th>STREET</th>
<th>POSTALCODE</th>
<th>CITY</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sara</td>
<td>Güttinger</td>
<td>Bahnhofstr. 18001</td>
<td>8001</td>
<td>Zürich</td>
<td>Switzerland</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14: Sample result of the query “address of sara”.

<table>
<thead>
<tr>
<th>FIRSTNAME</th>
<th>LASTNAME</th>
<th>STREET</th>
<th>POSTALCODE</th>
<th>CITY</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike</td>
<td>Escher</td>
<td>Taubengasse 223000</td>
<td>3000</td>
<td>Bern</td>
<td>Switzerland</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 15: Sample result of the query “address of mike”.

15
Both tables have the same relational key, even though they have different content. Therefore if a user gives a vote for the result of the first query, it also affects the result in the second query.

One problem that this key has, is that it is too general (i.e. its scope are all relations of all result sets). We have to test implications of that on further bigger data warehouses, to draw a conclusion, if and how we can adapt this key to make it more selective.

2.4 Scoring Formulas

The following section describes the formulas used to compute and combine the scores for the different user feedback methods. Note that in the following

\[ e_{..} = \text{count of votes of a certain type} \]
\[ s_{..} = \text{score of a certain feedback method} \]
\[ w_{..} = \text{weight for score or count} \]

2.4.1 Entry Point Specific Score Formulas

The entry point specific ranking is added to the query graph in the rank and top n phase of the system. In the following subsection, we explain how the score is computed and how the score is combined, with the already existing entry point ranking of the system. Note that in the following formulas

\[ e\text{pu}f = \text{entry point user feedback} \]
\[ li = \text{like} \]
\[ soso = \text{so so} \]
\[ wr = \text{wrong} \]
\[ \text{combep} = \text{combination of entry point specific scores} \]
\[ epsys = \text{entry point scoring computed by the search system} \]

The score of the entry point specific feedback is computed, by dividing the weighted count of positive votes by the weighted sum of all votes for the result. The weights can be chosen from the admin of the system. For our system we have taken the values 1 for the weights of the like and wrong votes and 0.5 for the weight of the so so votes. The result of the score is normalized so it lies between 0 and 1.

\[
s_{\text{epuf}} = \begin{cases} \frac{w_{li} \cdot c_{li} + w_{soso} \cdot c_{soso} + w_{wr} \cdot c_{wr}}{w_{li} + w_{soso} + w_{wr}}, & \text{if } c_{li} + c_{soso} + c_{wr} \neq 0 \\ 0, & \text{otherwise} \end{cases}
\]

(1)

To combine this score with the score the search system has computed for the specific query graph, we use the following formula. Since the system score is normalized, the combined score also lies between 0 and 1.

\[
s_{\text{combep}} = \frac{w_{\text{epsys}} \cdot s_{\text{epsys}} + w_{\text{epuf}} \cdot s_{\text{epuf}}}{w_{\text{epsys}} + w_{\text{epuf}}}
\]

(2)
The weight of the system score can be chosen by the system admin. We chose a value of 0.5.

The weight of the user feedback score depends on the number of votes given so far for the specific result. The formula shown below is 0, if no votes have been given so far, and increases with the number of votes, until they reach a max weight which can be defined by the system admin. Because of that, the score of a result is equal to the system computed score, if no user feedback votes have been given so far. It also helps to avoid too big jumps of the score when the first votes for a result are given, because the weight of the user feedback score reaches its max value only after a few votes.

\[ w_{epuf} = (1 - \left( \frac{1}{e_{Li} + c_{eso} + c_{Wr} + 1} \right)) \cdot \max W_{epuf} \] (3)

The standard value we chose for the maximal weight of the entry point user feedback is 1.

### 2.4.2 Relation Specific Score Formulas

The relation specific ranking score is added to the query graph in the SQL generation phase. In the following subsection it will be explained how the relational score for one relation and a whole result is computed. At the end it is shown how the the entry point specific score and the relation score are combined for one query graph. Note that in the following formulas

\[
rel = \text{relation specific} \\
pos = \text{positive} \\
neg = \text{negative} \\
combrel = \text{combination of relation scores} \\
combesp = \text{combination of entry point specific scores} \\
comb = \text{combination of all scores} \\
relvotes = \text{relation votes} \\
unvotedrel = \text{relations for which no votes have been given}
\]

To compute the score for one relation in a result, we use the weighted number of positive votes divided by the weighted sum of all votes of the relation. The resulting relation score is normalized so it lies between 0 and 1. Like in the entry point specific feedback the weights can again be chosen by the system admin. For our system, we took a weight of 1 for both the positive and the negative votes.

\[
s_{rel} = \begin{cases} 
\frac{w_{pos} \cdot c_{pos} + w_{neg} \cdot c_{neg}}{(w_{pos} \cdot c_{pos} + w_{neg} \cdot c_{neg})}, & \text{if } c_{pos} + c_{neg} \neq 0 \\
0, & \text{otherwise}
\end{cases}
\] (4)

To compute the score for one result we sum up the scores of all contained relations and divide it by the number of contained relations which have been rated.
The scores of the combined entry point ranking and the relation specific ranking are merged and added to the query graph in the SQL generation phase using the formula below.

$$s_{combrel} = \begin{cases} \sum_{\forall \text{relations} \in res_{rel}}^{s_{rel}} \frac{s_{rel}}{c_{ratedrel}}, & \text{if } c_{ratedrel} \neq 0 \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

The weight of the entry point specific score can again be chosen, the standard value which we use in our system is 1. The weight of the relational score, depends on the number of relations in the result, which have not been voted so far and also on the total number of relational votes for the current result.

$$w_{combrel} = (1 - \frac{1}{c_{relvotes} + 1}) \cdot \frac{\max W_{combrel}}{c_{unvotedrel}} \quad (7)$$

### 2.5 Summary Like Feedback

The Like feedback gives the system users the possibility to influence the scores of results. This increases the quality of the search system service, because the users take care of the result ordering. It also reduces the possibility that a good result is filtered out during the result finding process.

There are still more options to improve the system, e.g. integrate an authentication system so users can not vote as often as they want for one result. A solution for this problem and other improvements and enhancements are described in the future work section.
3 Graph Change/Proposal Feedback

3.1 Overview

The graph change and the graph proposal feedback allow the users to give feedback on the metadata graph of our search system. The graph change feedback can be used to enhance the metadata graph, with new keywords which were unknown before. This helps keeping the search system flexible and adaptable for changing demands of the users (e.g. new terminology or changes in the underlying data structure). The graph proposal feedback gives the user the possibility to vote for edges or nodes to be removed, if they are unnecessary, or propose new edges, if an edge is missing. This is helpful because parts of the metadata graph building process are automated (e.g. the building of the DBpedia part) and it is possible, especially in large search systems, that this process adds wrong edges and nodes or misses some edges that should be there. The votes of the users are collected and presented to the admin to help him keep the metadata clean. The difference between the graph change and the graph proposal feedback is that the graph change feedback is intended to enhance the metadata graph with new entry points. This can be used by users which are new to the system and that do not know the used terminology of the existing data warehouse. In contrast, the graph proposal feedback is intended for experienced users which know the underlying structure of the data warehouse to point out failures in the metadata graph and help the admin to keep it minimal and clean. Examples of the two feedback are given in sections 3.1.1 and 3.1.2. A precise description of the two feedback systems is given in sections 3.2 and 3.3.

3.1.1 Graph Change Feedback Examples

In the following section an example of how the graph change feedback works is given. Let’s assume the user has entered the query “stock buys of sara” on the search page. As you can see in Figure 16 the two keywords stock and buys are unknown to the system.

![Figure 16](image-url)

Figure 16: The top of the result page after the query “stock buys of sara”.

The user can add new nodes with the values of the unknown words to the metadata graph by clicking on the “+” button behind the unknowns. Figure 17 shows the result page after the two new nodes have been added to the metadata.
Figure 17: The top of the result page after the two new nodes are added.

With a click on the source link of a node a preview is opened which shows its subgraph of the metadata rooted at this node. Figure 18 shows the subgraph of the stock node.

![Stock Node Subgraph](image)

**Add Outgoing Edge**

![Add Edge](image)

**Propose edge to be removed / Remove Edge From New Node**

![Remove Edge](image)

**Propose node to be removed / Remove New Node**

![Remove Node](image)

Figure 18: The subgraph preview page of the new stock node.

You can see that the stock node has no edges because it was newly created. With the add outgoing edge drop-down the user can select an existing node to which the new edge is connected. We connect the new “stock” node with the existing “Securities” node (because stock is a synonym for securities). As you can see in Figure 19 the subgraph of the “stock” node is updated on the preview.
After also adding an edge from the “buys” node to the existing “Transactions” node and redoing the query “stock buys of sara” the system recognizes the two keywords and returns the result shown in Figure 20.

Figure 19: The subgraph preview page of the new stock node after adding a new edge.
Figure 20: The produced result after integrating the new nodes (“stock”, “buys”). Note that for space reasons a projection was set on the result.

It is also possible to add new nodes with values consisting of two words by separating the two words with a “-” (see Figure 21).

Figure 21: Example of a query with a two value keyword.

The “-” is replaced with a space when adding the new node to the metadata (see Figure 22).
Figure 22: Example of a query with a two value keyword after adding the two value unknown.

It is also possible for users to remove new nodes or outgoing edges from new nodes, using the remove new node or remove edge from new node drop-downs. This is helpful if a user finds out that he/she has accidentally created a new node or established a wrong connection between nodes and wants to undo it. However, this node and edge removal is only possible for new nodes and outgoing edges of new nodes. To inform the system about other wrong or unnecessary nodes or edges the graph proposal feedback has to be used.

3.1.2 Graph Proposal Feedback Examples

The following section shows an example of how the graph proposal feedback can be used to help the system administrator recognize unnecessary edges. Let’s assume an unnecessary edge was created from the DBpedia node “Country” to the node “Partners”. If a user now sends a query containing the keyword country, unnecessary and/or wrong results are produced. Figure 23 shows the result of the query “country of sara”.

Figure 23: Result of the query “country of sara”.

Input
buy transactions of sara

Output
buy transactions of [S]ara
Operator: Busness Object True
Value: Unknown

• buy transactions: userFeedback, USERFEEDBACK
• s: ali, UNKNOWN [2]
• s: Title: individuals.firstName, BASE_DATA

Score: 0.02 (0 / 0 / 0)

http://userfeedba/newNodes/buy_transactions ~ http://example-world.ch/bowietata/individuals/firstname/Sara ~ - -

23
Figure 23: Result of the query “country of sara”.

As you can see, both results contain the table “parties”, which is not necessary to find the country of a person. A click on the source link of the DBpedia node “Country” on the result page, brings us to the preview page of the metadata subgraph with root “Country” (see Figure 24). The preview shows us, that the DBpedia node “Country” has two outgoing edges, that connect it with the schema node “Country” and the DBpedia node “Partners”.

```sql
SELECT * FROM addresses, individuals, parties WHERE (individuals.firstName='Sara') AND (individuals.id=addresses.id) AND (individuals.id=parties.id) AND (individuals.firstName='Sara') AND (individuals.id=parties.id)
```

Result 1:
```
<table>
<thead>
<tr>
<th>ID STREET</th>
<th>POSTALCODE</th>
<th>CITY</th>
<th>COUNTRY</th>
<th>ID FIRSTNAME</th>
<th>LASTNAME</th>
<th>GENDER</th>
<th>BIRTHDATE</th>
<th>DEATHDATE</th>
<th>LIVINGATID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Bahnhofstr. 1 8001 Zurich Switzerland</td>
<td>S03 Sara</td>
<td>Guttinger w</td>
<td>1952-04-23</td>
<td>1999-01-01</td>
<td>2 S03</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Figure 24: Preview page of the metadata subgraph with root “country”.

To tell the system, that the outgoing edge to the “Partners” node is wrong, we can use the propose edge to be removed drop-down on the preview page. Every edge has a remove proposal counter, which is raised with every proposal for it and shown to the administrator in the admin tool (see Figure 25).

![Propose Outgoing Edge](image)

Propose edge to be removed / Remove Edge From New Node

![Propose node to be removed / Remove New Node](image)

Figure 25: Part of the admin tool showing the DBpedia “Country” node and its edges.

It is also possible for the user to propose the removal of nodes, by the propose node to be removed drop-down seen in Figure 24. Similar to the edges, also all nodes have a remove proposal counter, which is incremented with every vote and presented to the admin in the admin tool (see Figure 25). The system also allows a user, to propose new outgoing edges from existing nodes. This can be done using the propose outgoing edge drop-down on the metadata subgraph page of a node (see Figure 24). The edge proposals are stored persistently and presented to the admin using a table in the admin tool. Figure 26 shows such a table for the Country node, where one outgoing edge is proposed.

![Country and Partners](image)
3.2 Graph Change Feedback

3.2.1 Description

The graph change feedback allows a user to enhance the metadata without interaction of the system administrator. This enables the system to learn new keywords from its users. The feedback system introduces a new node type (User Feedback type) to the metadata graph, to have a clear separation between the old core types and the nodes generated by the users. The graph changes are only done in memory, and would be lost after a restart of the system. To prevent that, the graph changes performed by the users are stored persistently and in chronological order in a file. While booting the system, this file is parsed and all changes are executed again on the graph. The system does not allow its users to perform changes on the core metadata graph (e.g. remove schema nodes, remove edges between core nodes . . . ), because this would allow malicious users to sabotage the running search system. To give the users the possibility to give feedback on the core part of the metadata, we designed the graph proposal feedback system (see section 3.3).

3.2.2 User Possibilities

This section describes the possibilities of the graph change feedback.

Add new nodes: The user can create new entry points to the metadata graph by adding new nodes to the graph. As described in section 1.2.1, the system classifies keywords which are not identified as business objects, operators or values as unknowns. With the graph change feedback, the user can now pick such an unknown and create a new node with a value equal to it. The previously unknown value will now be identified by the system.

Delete new nodes: Since it might happen that a node is created accidentally, the graph change system also allows the users to delete new nodes. Note that the system only deletes nodes with type “User Feedback”. For all other node types the user can just send a remove proposal using the graph proposal feedback system.

Add outgoing edges to new nodes: Of course a new entry point which has no connections to other nodes is useless. Therefore, the graph change system also provides the possibility to connect the new nodes with the rest of the graph by creating outgoing edges for them. After the new node is correctly integrated into the metadata graph, the search system can recognize the keyword which it represents and produce the right solution.

Figure 26: Part of the admin tool showing the outgoing edge proposals for the “Country” node.
Delete outgoing edges from new nodes: It might happen that a user accidentally adds a wrong edge to a new node. To undo such a mistake, the graph change system also allows the users to remove outgoing edges from new nodes.

3.3 Graph Proposal Feedback

3.3.1 Description

The graph proposal feedback allows the system users to give feedback on the metadata graph without changing it. The gathered feedback is stored persistently and is accessible by the system administrator. The administrator can use this information to optimize the metadata graph and adapt it accordingly. The graph proposal feedback allows the system user only to propose changes on the metadata graph and not perform them directly (e.g., propose to remove an edge). The reason for this is to protect the metadata graph from changes that are done by malicious or unqualified users.

3.3.2 User Possibilities

This section describes the possibilities of the graph proposal feedback.

Propose to remove a node: The user can propose to remove a node and all its incoming and outgoing edges from the metadata graph. The feedback system stores a remove proposal counter for every node in the metadata graph. The counter for a node is incremented every time a user gives a removal proposal for it. The administrator can look up the counter for a node in the admin tool and decide if the node is needed or not.

Propose to remove an edge: The user can also propose to remove single edges from the metadata graph. Similar to nodes, the feedback system stores a remove proposal counter for every edge of the metadata graph. This counter is made available to the administrator in the admin tool, where he/she can decide if the edge should be deleted.

Propose new edges: The graph proposal system also offers the possible to propose new edges for the metadata graph. The proposed edges are stored by the feedback system, together with a counter of propose votes for them. The proposed edges are presented together with their vote count to the administrator, who can decide if the new edge makes sense or not.

3.4 Summary Graph Change/Proposal Feedback

The graph change feedback allows the search system users to easily enhance the metadata. This is done, by giving them the possibility to create new entry points to the metadata, by adding new nodes and connecting them to the existing ones. The system does not allow to directly change or modify parts of the core metadata graph (DBpedia, domain ontology, schemas), because that would make the system vulnerable for attacks of malicious user which could corrupt the metadata graph by deleting nodes or inserting wrong edges. The graph proposal feedback bridges this gap, by giving the (more advanced) users the possibility to make proposals on how to improve the core of the metadata graph. The combination of both techniques make the whole search system more flexible,
because they allow slight changes on the metadata at runtime and also help the system administrator to optimize and enhance the core of the metadata.
4 Projection/Order By Feedback

4.1 Overview

The search system, as it is, only returns “Select * ...” queries as results. Because of that, most results consist of many unnecessary columns, which make the result far more complex than it has to be. There is also no possibility to set the column order in the relation or the order by criteria of the result tuples. The projection and order by feedback changes this by giving the users the possibility, to set the visibility and order of columns in a result relation. It also allows the users to order the relation tuples, by declaring the columns after which they should be ordered. This simplifies the evaluation of the results and increases their quality.

4.1.1 Examples

Imagine the user issued the keywords “address of Sara”. As you can see in Figure 27, the search system returns two results. We see that the relations of the results contain unnecessary columns (e.g. the Id columns) and that the columns have no special order. We can now change the projection and order by criteria of the relation containing the address data, by clicking on the “->” button right under it in the first result. The Button leads us to the projection configuration page shown in Figure 28.
Result [0]:
Average Score: 0.73
SELECT *
FROM addresses, individuals, parties
WHERE
    (individuals.firstName='Sara') AND
    (individuals.id=parties.id) AND
    (individuals.livingAt=addresses.id)
Relation Feedback: [1 / 1]  
ID | STREET      | POSTALCODE | CITY     | COUNTRY | ID | FIRSTNAME | LASTNAME | GENDER | BIRTHDATE | DEATHDATE | LIVINGAT | ID
---|-------------|------------|---------|---------|----|-----------|----------|--------|-----------|-----------|----------|----
 1 | Bahnhofstr. 1 | 80001      | Zürich  | Switzerland | 503 | Sara       | Guttinger | w      | 1982-04-23 | 1999-01-01 | 2         | 503

Change Projection: [1]
Result Feedback: [1] | Ok | No | Wrong

Result [1, 2]:
Average Score: 0.375
SELECT *
FROM addresses, individuals, parties
WHERE
    (individuals.firstName='Sara') AND
    (individuals.id=parties.id) AND
    (individuals.livingAt=addresses.id)
SELECT *
FROM organizations, parties
WHERE (organizations.id=parties.id) (produced 2 times.)
Relation Feedback: [1 / 1]  
ID | STREET      | POSTALCODE | CITY     | COUNTRY | ID | FIRSTNAME | LASTNAME | GENDER | BIRTHDATE | DEATHDATE | LIVINGAT | ID
---|-------------|------------|---------|---------|----|-----------|----------|--------|-----------|-----------|----------|----
 1 | Bahnhofstr. 1 | 80001      | Zürich  | Switzerland | 503 | Sara       | Guttinger | w      | 1982-04-23 | 1999-01-01 | 2         | 503

Change Projection: [1]
Relation Feedback: [1 / 1]  
ID | COMPANYNAME | ESTABLISHEDATE | CLOSINGDATE | ID
---|-------------|-----------------|-------------|----
 1 | Pfaer       | 1849-01-01      | 1999-12-31  | 481
 2 | Novartis    | 1995-12-20      | 9999-12-31  | 402
 3 | Latham      | 1950-01-01      | 2008-09-15  | 403
 4 | CS          | 1856-07-05      | 9999-12-31  | 404
 5 | ETH         | 1855-01-01      | 9999-12-31  | 405

Change Projection: [1]
Result Feedback: [1] | Ok | No | Wrong

Figure 27: The result page for the query “address of Sara”. Note that this Figure is equal to Figure 7 its repeated for better readability.
Relation Projection Page

For Relation: [addresses, individuals, parties]

<table>
<thead>
<tr>
<th>Visible Column</th>
<th>Priority</th>
<th>Order</th>
<th>By Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>parties</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>addresses</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>postcode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>street</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>country</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>city</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>individuals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>lastname</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>living</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gender</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>amba</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>firstname</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>deathDate</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Preview

```
SELECT * FROM addresses, individuals, parties
WHERE
  ((individuals.firstname='Sara')) AND
  (individuals.id=parties.id) AND
  (individuals.living=addresses.id)
```

<table>
<thead>
<tr>
<th>ID</th>
<th>STREET</th>
<th>POSTALCODE</th>
<th>CITY</th>
<th>COUNTRY</th>
<th>ID</th>
<th>FIRSTNAME</th>
<th>LASTNAME</th>
<th>GENDER</th>
<th>AMBAL</th>
<th>DODATE</th>
<th>BIRTHDATE</th>
<th>DEATHDATE</th>
<th>LIVINGAT</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 Bahnhofstr. 1 8001</td>
<td>Zurich Switzerland</td>
<td>503</td>
<td>Sara</td>
<td>1962-04-23</td>
<td>9999</td>
<td>01-01</td>
<td>503</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 28: The projection configuration page for the relation with key [addresses, individuals, parties].

As you can see the configuration page gives us the possibility to define the visibility, the order in the relation and the order by priority for a column. After entering the selections the user can click on the “updatePreview” button to see how the relation would look with the entered configurations see Figure 29.
Relation Projection Page

For Relation: [addresses, individuals, parties]

<table>
<thead>
<tr>
<th>Visible Column</th>
<th>Priority</th>
<th>Order By</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>parties</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addresses</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>postcode</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>street</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>country</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>city</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>individuals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lastname</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>livingAt</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gender</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>birthDate</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>firstname</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deathDate</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Preview

```
SELECT individuals.firstname, individuals.lastname, addresses.street, addresses.postalcode, addresses.city, addresses.country
FROM addresses, individuals, parties
WHERE
    (individuals.firstname = 'Sara') AND
    (individuals.id = parties.id) AND
    (individuals.livingAt = addresses.id)
ORDER BY individuals.firstname, individuals.lastname
```

<table>
<thead>
<tr>
<th>FIRSTNAME-LASTNAME</th>
<th>STREET</th>
<th>POSTALCODE</th>
<th>CITY</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sara</td>
<td>Guttinger</td>
<td>Dahnhofstr. 1801</td>
<td>Zürich/Switzerland</td>
<td></td>
</tr>
</tbody>
</table>

Figure 29: The projection configuration page for the relation with key [addresses, individuals, parties] after an update of the preview.

With a click on the “reset” button, the system resets the user selections on the configuration page to the old values. A click on the “confirm” button stores the selections and brings the user back to the result page. In Figure 30 the results after the made selections are shown. In this case, the projection and order by criteria is used for both results.
Figure 30: The results of the query “address of sara” after the projection selections.

The reason for this is, that no user has set a projection for the relation in this result before, so the projection which most users picked is employed. If we know change the projection of the relation in the second result (e.g. show a additional column), it only changes the appearance of the relation in the second result. The first result is unchanged, since the user has set a projection for this result before (see Figure 31).
Figure 31: The results of the query “address of sara” after the projection of the relation in the second result was changed.

The projection and order by configurations also affect relations in other result sets for which the user has not set a projection jet, e.g. the result of the queries “address of mike” or “show all last names” (see Figure 32).
For the second query we see that the result is still not optimal, so we can change its projection to show only the “LASTNAME” column and also order the result tuples by it. Figure 33 shows the adapted result.

This change has no influence on the previously configured relations, i.e. the two results for the query “address of Sara” still look as in Figure 31.

4.2 Description

The projection/order by feedback system consists of three maps. Two of these maps are stored persistently by the feedback system and map a key, which identifies a relation, to a list of projection/order by configurations. Each configuration is coupled with a vote counter which is increased, when a user picks the specific configuration for the relation with that key (see Figure 34).
The configuration lists are sorted by these vote counters, so that the configuration with the most counts is at the first place. Both persistent maps have different keys. The first key is a concatenation of the key used in the entry point specific feedback to identify one result (see section 2.2.2) and the key used in the relational feedback (see section 2.3.2). The key of the second map is equal to the key used in the relational feedback, namely it is a string representing the join set of a relation (i.e. the names of the tables in the database which form the relation). When the system looks up a configuration for the projection and order by criteria of a result relation, the first map has a higher priority because it represents the configurations made for relations in the context of a result. If no entry is found in the first map, the feedback system searches for a configuration in the second map. The configurations in the second map are stored, according to the join set of their corresponding relation without the result context. If again no configuration is found, the result is presented as before with a “Select * . . . ” statement. Figure 35, shows an example of the second persistent map with the join set of the relation as key. The first config object in the list has a vote count of 1 and stores for all columns of the relation its name, visibility, column order and order by priority.

![Figure 34](image-url)

**Figure 34:** A schematic example of a projection map.

With only the two maps described above, the projection system would always show the most selected configuration for a result relation. The problem of this is that a user, if he/she changes the projection of a relation, wants to use his/her projection further on for this relation. To realize that, a map which stores all projection/order by configurations made by a user is stored in the users HTTP session. The key of this map, is equal to the key of the first persistent map (i.e a combination of the entry point specific and the relational key). The key is mapped to the last configuration, the user picked for the relation represented by it (see Figure 36).

![Figure 35](image-url)

**Figure 35:** A example of the second projection map with a projection/order by configuration.

With only the two maps described above, the projection system would always show the most selected configuration for a result relation. The problem of this is that a user, if he/she changes the projection of a relation, wants to use his/her projection further on for this relation. To realize that, a map which stores all projection/order by configurations made by a user is stored in the users HTTP session. The key of this map, is equal to the key of the first persistent map (i.e a combination of the entry point specific and the relational key). The key is mapped to the last configuration, the user picked for the relation represented by it (see Figure 36).
This map has the highest priority, so if the user changes a configuration for a relation, the system uses this configuration instead of the one with the highest votes. Figure 37, shows an example of the persistence map stored in the session. As you can see, the key is a combination of the key used in the entry point specific feedback and the join set of the actual relation. This example is a snapshot from the map after our previous example with the query “show all last names” where we set the visibility of all columns except the “LASTNAME” column to false.

Of course to store the map in the session has the disadvantage, that it is not persistent and lost if the user ends his/her session. So if he/she starts a new session, the previous configurations are lost and again the ones with the highest votes are picked. This problem could be solved by adding a authentication system to the search system. We will discuss that in the future work section.

4.3 Summary Projection Feedback

Through the possibility to adapt the projection and order by criteria, the feedback system increases the quality of the returned results. The selections made by a user, not only affect his/her own results, but also results of other users (e.g. for relations which the current user has not set a projection/order by configuration yet, the configuration which is picked by most users is taken by the system). This also makes it easier for new users, to browse through results and evaluate them. With the addition of a authentication system and the use of some techniques described in [3], we can further enhance this feedback system. This enhancements will be discussed in the future work section.
5 Persistence

The changes of our feedback systems are applied in main-memory. That means
that they would all be lost after a restart of the search system. To make the
changes persistent we store them in XML files, that are parsed when booting
the search system. In the following subsections, the structure of the XML
persistence files for every feedback type will be briefly explained.

5.1 Like Feedback

Figure 38: The format of the like persistence file.

Figure 38 shows the format of the entry point specific feedback persistence file. 
As described in section 2.2, this feedback holds a table which stores the vote
counts of the users. To store this table persistently, for every row in the table a
XML “feedback” object is created, which contains the key and the vote counts.
The format of the relational feedback file looks basically the same, but it only
stores two vote counts (positive, negative) in every “feedback” element.

5.2 Graph Change Feedback

Figure 39: The format of the graph change persistence file.

As seen in Figure 39, the graph change feedback stores a “feedback” element for
every change done to the metadata graph. Each element has three attributes
(type, sourceNode, destinationNode). The type attribute, indicates the type
of change done to the graph (e.g. add an edge, remove a node ...). The
sourceNode and destinationNode attributes carry the information needed to
perform the change, for example the source links of the nodes connected with a
new edge (note that if only one of the attributes is needed e.g. in the add node
case both attributes are set with the same value).
5.3 Graph Proposal Feedback

The graph proposal feedback stores a “proposal” XML elements which represent the given user feedback. As shown in Figure 40, the proposal element has an attribute which indicates the type of given proposal (e.g. proposeEdge, proposeEdgeToBeRemoved, ...) and several subelements, which provide the information needed for the proposal. A element of type “proposeEdge” or “proposeEdgeToBeRemoved” always contains one “from” element, which holds the identifier of the edges source node. Further more it contains one or more “to” elements (one for each proposed edge with source node equals to the node in the “from” element), that specify the destination nodes of the edges. Each “to” element holds a “count” attribute, representing the number of proposals for that edge. A proposal of type “proposeNodeToBeRemoved” only contains one subelement, which holds the specific node source link and the count of remove votes.

Figure 40: The format of the graph proposal persistence file.
5.4 Projection/Order By Feedback

The format of the projection persistence file is shown in Figure 41. The system creates a projection XML element for each projection/order by configuration, stored in the two persistent projections map explained in section 4.2. Each projection element has three attributes, the type of the feedback (specific if projection is part of the first hash map, global if projection is part of the second hash map), the id (key) under which the projection is stored in the map and its vote count. For each column of the relation, the projection/order by configuration represents, the projection element contains a “column” subelement. Each “column” element contains four attributes which hold its id (table name + column name) and its projection and order by configuration (visibility, column order, order by priority).
6 Testing

The testing of a feedback system like the ones described in this paper consumes a lot of time and resources. Because of that we were not able to actually test our feedback system in a real world scenario until now. To give you an impression on how such a test could look like for our system and what results we expect in an optimal case, a test scenario is described in the following section.

We propose to set up the search system on several (5-10) big data warehouses (all with different data and a different data structure), which are images of existing data warehouses operated in companies. All systems also have a system admin, that knows the data warehouse and is able to perform changes on the metadata. Every search system is used by a group of people, that has already worked on the data warehouse in their company and some people which are completely new to the system. All users are shortly briefed in how to use the search system and the contained feedback features. To test the system, scenarios are designed, which resemble the normal work on the specific data warehouses. After all scenarios have been worked through the users of the systems are completely exchanged by new ones and the system scenarios are repeated, to see if the feedback has improved the usability of the search system. During the test every system interaction together with the id of the specific user is logged. With that we can analyze the results also according to types of users (i.e. experienced users vs. new users). After the test also a questionnaire is given to all system users, where they can rate the usability of the feedback features and give comments to the system. In the following, it will be described how the test results for the several feedback features should look like in an optimal case.

Like feedback: The like feedback system is used to influence the order of the presented results so that the top result is at the first place. In the optimal case the test would show, that the like feedback is heavily used at the beginning of the test and after a short “warm up” phase the usage goes rapidly down. The usage should also not increase too much then the user group is exchanged, because that would indicate, that the users have adapted to the system and not vice versa. The optimal behavior would show, that the users used the like feedback to order the returned results at the “warm up” phase of the system and afterwards, the system reaches a state where the results of the most common queries are correctly sorted (see Figure 42). If an increased usage of the like feedback is noticeable for some results, it might indicate that there exist several groups with different demands among the users of one data warehouse. This problem can be handled by either adapt the feedback system (if it allows grouping of users) or split the search system of one data warehouse. The reason for the second is, that every subgroup among the test users work on their own system and has its own feedback.
Graph Change Feedback: The graph change feedback provides the system users, the possibility to enhance the metadata of the search system with new entry points. The expected results should show, that this feature is more used by the part of the users, which are inexperienced with the data warehouse. The reason for that is, that the experienced users are familiar to the special terms used in the data warehouse, which are stored in the domain ontologies of the search system, and directly use them in their queries. In contrast to that, inexperienced users do not know these terms and use commonly used synonyms, which the search system might not recognize from the beginning. In an optimal case, the usage of the graph change system should also go down after a while and the system should reach a stable state. The change of the user group could raise the use of the feedback system again, but it should flatten and fall again after a short warming phase (see Figure 43).

Graph Proposal Feedback: The graph proposal feedback is used, to provide
advanced system users the possibility, to give feedback on the metadata of their search system. Because it is intended for more advanced users, the test should show that it is used by the users, which already have experience with the data warehouse (and its data structure). We expect a usage peak of this feedback system at the beginning, because the users will use it to correct errors in the metadata, which are introduced by the automatic metadata graph generation of the search system. If the search system admin uses the proposals correctly, the metadata should reach a stable state and the proposal feedback should not be used anymore in the further running of the system. Also the change of the user groups should not raise the use of this feedback again, because the metadata should have reached its optimal state at this point (see Figure 44).

Figure 44: Expected usage curve for the graph proposal feedback system.

**Projection Feedback:** The projection feedback should also heavily be used at the beginning of the test. After a while, when the projections and order by configuration of the most queried result relations have been set, the usage of this feedback system should go down (see Figure 45). If the test shows that several relations have a constant usage of the projection feedback system during the test this might have the same reason as described above in the like feedback part.
Figure 45: Expected usage curve for the projection feedback system.
7 Related Work

In the following section I will give examples of related projects like other keyword based search systems, user feedback systems or papers which describe methods to improve them. I will also mention papers which propose methods that could be used to improve or enhance the feedback features we added to our system.

7.1 Search Systems

DBXPLORER [2]: DBXPLORER is a system that enables keyword based searches on relational databases. Instead of the metadata graph used in the system described in section 1.2.1 it uses the so called “Symbol Table”, to look up where the searched keywords are stored in the database. The paper also introduces different designs and compression methods for the symbol table and shows their advantages and disadvantages. DBXPLORER takes a set of keywords and looks them up in the symbol table to find the rows in the relational data that contain them. Only rows which contain all keywords from the user query are considered as correct result by the system. The system operates in two steps Publish and Search. In the Publish step the database along with the set of tables and columns which should be made available for keyword search are marked and the symbol table is built. In the Search step the symbol table is looked up for all places where the keywords appear and SQL statements are created, which are ranked and presented to the user.

DISCOVER [9]: DISCOVER operates on relational databases allowing users to issue keyword queries. It returns sets of tuples that are associated because they join on a primary key foreign key relationship and collectively contain all the keywords of the query. To do the matching of keywords from the user query to relational tuples DISCOVER uses a “Master Index” which stores the keywords contained in each tuple. The system proceeds in the following steps. First the set of keywords given by the user is looked up in the “Master Index”, which returns the tuple sets for each relation which contain the keywords. Then Discover calculates all candidate networks, i.e., join the expressions on primary key to foreign key relationships. Because the candidate networks share join expressions, immediate results can be computed and reused for computation of multiple candidate networks. The next step is to produce an execution plan, that calculates and uses intermediate results. Finally an SQL statement is produced for each line of the execution plan and passed to the DBMS. The difference between DISCOVER and DBXPLORER is that DBXPLORER does not consider solutions that include two tuples from the same relation and it only considers exact matches i.e. a keyword must match exactly an attribute value. Furthermore DBXPLORER does not exploit the reusability opportunities of the join trees. The difference between DISCOVER and the search system we use is that in our system a metadata graph is used to bridge the gap between the keyword queries and the relational data. In contrast to that DISCOVER uses a similar approach as DBXPLORER to find the matching tuples in the relational data. Both use an index which maps keywords to the locations where they are contained in the relational data.
PRECIS [11]: The paper explains the notion of precis queries. A precis query is a keyword based query which takes unstructured keywords and returns structured answers. In detail the result is a database which is a logical subset of the original one (i.e. it contains not only items directly related to the keywords but also items implicitly related to them). The advantage of such a precis query is that it combines the advantages of both the structured and the unstructured world. A user does not need knowledge about the underlying schema to query the system and the result being a database has the advantage that it can also be queried for further refinement, reviewing or restructuring of the data. The difference between this and earlier work is that earlier work was restricted to precis queries with a single keyword. They introduce logical operators (AND, OR, NOT) which allow users to write queries of several combined keywords (e.g. "Clint Eastwood" AND "thriller").

EFFICIENT IR-STYLE [8]: The Goal of this paper is to enable IR-style keyword search over a db without knowledge of the schema, the query language or the role of the keywords. The problem they try to solve is that IR keyword style is well developed for document search but in databases information can be spread over several tables which are connected through primary key foreign key relationships. For that reason they do not only search keywords in the same tuple or the same relation but also in tuples connected through a primary key foreign key relationship. The scoring of the result is done via the keywords distance to each other in the solution and an IR-style score of the result tree. The result of a query is a tree where each edge corresponds to a primary key foreign key relationship and no tuple is redundant (which means the solution is minimal). Also the tree either contains all query keywords (AND semantics) or at least one query keyword (OR semantics). The advantage of this solution is that in contrast to DBXPLORER and DISCOVER it uses well studied information retrieval ranking strategies and extends them to databases. Apart from that it also supports AND and OR semantics as described above.

BLINKS [7]: BLINKS allows top k keyword searches on schema less node-labeled Graphs. A top k keyword search finds the top k answers according to some ranking criteria, where each answer is a substructure of the Graph containing all query keywords. BLINKS follows a search strategy with provable performance bounds and exploits a bi-level index for improving the search. Keyword search in graphs can be applied in various fields such as browsing XML docs, techniques for the Semantic Web (OWL, RDF) and keyword search solutions over relational databases which treat tuples as graph nodes connected via foreign key relationships. BLINKS partitions a data graph into multiple subgraphs, or blocks. A bi-level index consists of a top level block index, which maps keywords to blocks, and an intra block index which holds more detailed information about a block.

BANKS [5]: BANKS is an example of a keyword based search system over databases that treats the relational tuples as nodes of a graph. This system enables keyword based search on relational databases, together with data and schema browsing. A user can get information by typing a few keywords, following hyperlinks and interacting with controls displayed on the results. As described above BANKS models tuples as nodes in a Graph connected by links.
which indicate an public key foreign key relationship. The results of a query are modeled as rooted trees connecting tuples that match individual keywords. The difference between BANKS and our search system is, that BANKS models the relational data as tree. The search system we use builds a tree using metadata (e.g. ontologies, database schemas). The nodes of the metadata tree in our system, contain references to the relational data to find the matching results.

7.2 Search System Extensions

DBEASE [10]: The authors of this paper propose a search as you type functionality to improve existing database search systems. This technique can be used to improve several types of search systems over databases for example keyword based, form based and SQL based search systems. Every time the user changes something on his/her query the results are adapted automatically on the fly and presented to him/her. The advantage if this technique is that the user immediately sees the result of his/her search and can adapt the query on the fly until the result matches his/her expectations. To illustrate the advantage of this technique the paper introduces a prototype database search system called DBEASE which uses the described technique.

The Potential of User Feedback through Iterative Refining of Queries in an Image Retrieval System [4]: The paper describes the problem of ambiguous user queries, which lead to poor results in keyword based search systems. To solve this problem, they propose the use of an iterative user feedback system, which is integrated into a keyword based image retrieval system to demonstrate its functionality. The feedback system, takes a keyword query from the user and returns categories of images for the user to choose. According to the selected category, the query of the user is adapted and again evaluated. This process is repeated until the search results match the expectations of the user. The proposed system is a classical example of a relevance feedback system, where during a conversation of the user and the system the query is refined to improve the results.

Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions [1]: This paper starts with an overview of state-of-the-Art (2005) recommender systems and describes their functionality. It also describes the limitations of these systems and proposes ways to improve them. Recommender systems are used to provide the users of a system with personalized recommendations and advertisements e.g. books in a bookstore or banners on websites. The problem that recommender systems solve, is to estimate a (good) rating for an item which has not been rated by the user so far. The paper mentions 3 types of recommender systems:

- Content-based systems: A content-based system recommends Items which are similar to the ones the user preferred in the past. The paper explains a commonly used method for calculating the similarity of two documents using the words contained in them and their frequency. One drawback this method has is that the contend of the element to rate has to be either
automatically parseable (e.g. text) or content features have to be assigned manually. Another problem mentioned is called overspecialization. The user only gets recommendations for items which are similar to the ones he/she voted for. Because of that no other items are introduced to the system user which might be very bad. Another drawback is that new users first have to rate a sufficient number of items until the recommendation system works properly.

- Collaborative systems: A collaborative system calculates the similarity of users. The rating of an item which has not been rated by a user is then predicted by looking at the ratings of users which are similar to the one we are looking at. The paper describes commonly used formulas to compute the similarity of users and a unknown ranking given these similarities and the rankings of the other users. These systems also have the drawback that new users first have to rate sufficiently many items before the recommendation works properly for them. Apart from that also new Items need to be rated by sufficiently many members before they can be recommended by the system. Another problem of such systems is that they suffer under the sparsity of the given feedback which leads to bad recommendations. Examples for that are:
  
  - some users are not similar to any other users
  - only a small fraction of users give ratings for items

- Hybrid systems: Most recommender systems, practically used, are hybrid systems, which means they are a combination of content-based and collaborative systems. The reason for this is to overcome the limitations of content-based and collaborative systems as they are mentioned above.

7.3 User Feedback Improvements

User Feedback as a First Class Citizen [3]: Some good improvements for user feedback are presented in this paper. It introduces several possibilities to maximize the benefits of user feedback in information integration systems. They define feedback as a 4 tuple \(<\text{obj}, \text{t}, \text{u}, \text{k}>\) where obj is the object for which the feedback is given, t is the term of feedback, u is the user that gave the feedback and k is the kind of feedback (e.g. \(<(t1,t2), \text{before}, \text{tim}, \text{ranking}>\) is an example for a ranking feedback of user tim saying that \(t1\) comes before \(t2\)). They also define that two feedback f1 and f2 are conflicting if \(f1.\text{obj} = f2.\text{obj}, f1.\text{u} = f2.\text{u} \text { and } f1.\text{k} = f2.\text{k}. To take care of inconsistency in feedback and/or changing demands on the system the paper proposes that in the case of conflicting feedback only the freshest is declared as valid and used for feedback. The paper also proposes a way to compute similarity of users by comparing their given feedback in terms of what they like and don’t like. This similarity measure can be used to build clusters of users and find out if a system needs to be split (because there exist groups with concurrent interests) or if two systems could be merged (because the users of the systems have similar interests). Furthermore the paper proposes a way of computing the expected feedback of a user for an object by looking at the feedback of the other users and their similarity to the user.
7.4 Similarity Concepts

Query Similarity [6]: The authors of the paper propose a technique which projects the Query-Flow Graph into a geometric space and then uses the resulting space to measure the similarity of two queries. The Nodes in the Query-Flow Graph represent different queries. Two nodes are connected with an edge if the corresponding queries are likely to appear as part of the same search goal. The edges are weighted by their likelihood so the higher the node edge weight the higher the similarity of two nodes. To build the flow graph for a system query logs are needed which document the queries of users over a significantly large time space. The projection method they propose is called spectral projection which projects the graph into an Euclidean space. The advantage of this method is that it is general and can be also applied to other graphs generated from query logs.
8 Future Work

The following section describes techniques to improve our feedback system and also introduces feedback features we were not able to implement because of time constrains.

The biggest problem of the feedback features we have, is that the users can give as many votes as they want on one result and they are all used for the final score. The reason for this is that the search system currently does not have an authentication system. Therefore the feedback system does not have access to user information. After an authentication system is integrated, we can use it for example in the voting process of the like feedback. Every vote could be stored together with the id of the user that gave it. If a user votes two times for a result with same id, the older vote is deleted and the new one is stored instead (same can be done with relational feedback votes). With this the whole system gets more flexible in terms of changing user requirements. To give an example, if the requirements of a user change over time and its old feedback does not match his/her demands anymore he/she can simply replace his/her old feedback by voting again.

Having an authentication system, we could also use the collected feedback (in combination with the user ids), to calculate a similarity measure between system users. This measure can be used, to cluster the users into groups with a big similarity. With that we can adapt the feedback system, so that given feedback only affects the results of the users in the same user group. This would solve the problem of concurring subgroups with different demands among the users of the search system (as described in section 6, for the like and the projection feedback). Because the clustering process can be automated and repeated from time to time, the system also takes care of changes in the user groups. Both of these improvements are introduced and described by Belhajjame et. al. in [3].

A problem of the graph change feedback, is that it is possible to add stop words like “from” or “to” to the metadata. If a malicious user does that and connects this new nodes with existing nodes, all queries containing these stop words produce wrong results. One possible solution to prevent this, is to introduce a new keyword type (e.g. stop word) to the search systems lookup phase. With that, the system could distinguish between unknown keywords and stop words. Another solution would be, to introduce a scope for the changes so that changes on the metadata of one user only affect his/her results. The problem of this is, that it is a quite big change in the search system and we would lose the positive effect of “good” graph change feedback on the results for all system users.

As mentioned in section 2.3.2, we might also get problems when using the relation feedback on bigger data warehouses, because the used key is to general. If this is the case, we have to enlarge the key by a part which reduces its scope (e.g. to the scope of similar result sets). As an example to reduce the scope to all result sets of “similar” user queries, we could use the terms in the query (without the unknowns and stop words) and create an index, which maps these terms to a key (e.g. a unique number). This key could then be used to enhance the key for the relational feedback to give it a scope that spans similar result sets.

A feedback feature, which we also would like to implement, is the “Result Proposal Feedback”. The purpose of this feedback is, to give the user the
possibility to enter the correct SQL statement for his/her keyword query, if none of the presented results matches his/her expectations. The so gathered proposals will be stored persistently and can be recommended to other users, that query the same or a similar queries. The proposed feature is basically a recommender system so we could use techniques used for other recommender systems, e.g. the similarity measures described in [1] or [6].
9 Conclusions

In this report we added different user feedback features to a keyword based search system over data warehouses. The added features give the user the possibility to

- Influence the score of results (Like Feedback)
- Enhance the metadata graph with new Nodes (Graph Change Feedback)
- Propose Changes on the metadata graph (Graph Proposal Feedback)
- Change the projection and order by criteria of relations (Projection Feedback)

We have shown with several examples, that our user feedback extensions increase the service quality, flexibility and usability of our search system. We also gave proposals for further enhancements which solve the problem of changing user requirements or the problem of concurring groups among the users of a search system.

The next steps will be to implement the additional enhancements and to test the improved search system in a real world scenario. We hope that this report gave you a good impression on how user feedback can be used to enhance a search system and what its advantages compared to other solutions are.
References


