SEMESTER THESIS SS06

Ranking & Scoring in iMeMex

Roger Jäggi
jaeggir@student.ethz.ch

Supervising Assistant: Shant Kirakos Karakashian
Supervisor: Dr. Jens-Peter Dittrich

September 20, 2006
ETH Zurich

Contents

1 Introduction

1.1 iMeMex Data Model

It is necessary to understand the underlying iMeMex data model to see how ranking and scoring works. This sub section gives a short introduction, more information is available under [1] and [2].

In the iMeMex data model (iDM), an information unit is called a **resource view**. All information in the system is exposed through resource views. iDM does not differentiate between files and folders. Even parts of files (as for example XML nodes) are mapped to resource views. Therefore, there is no longer a boundary between a file and its content.

A resource view consists of four parts, called components:

- **Name component**
  A finite string to represents the name of a resource view.
- **Tuple component**
  A list of attribute value pairs. A pair consists of the name and the value of an attribute. There are attribute classes for predefined resource view types.
- **Content component**
  The content of a resource view. This is usually text or binary data.
- **Group component**
  References to other resource views. For folders, it contains references to all files and sub folders in it.

These components are mapped to Lucene indexes. If the users executes a query, the query processor parses the query and executes it on some of the indexes depending on the query type.

1.2 The Scoring Scheme

The scoring scheme is responsible to rank the elements in the result set of a query. It is done with rules. Each rule computes an independent scoring value for a given resource view. The higher the score, the better the resource view matches the query. The sum of all these scores is used afterwards by a ranking operator to display a sorted result to the user.

The scoring scheme performs the following steps to execute a query and showing the result to the user:

0. Register all scoring rules. This is done only once.
1. Parse & execute query. Returns an unordered result set.
2. Apply all rules to all elements in the result set.

3. Compute total score for each resource view.

4. Sort the resource views by total score.

5. Display sorted result.

Step zero is done only once. On startup, the system registers all rules in the scoring interface. Afterwards, the scoring scheme can use the interface and its rules for all queries.

Ranking of web search engines like Google is based on PageRank, tfidf, proximity (distance between the terms) and the position of a term in a document (title tag, headline, subtitle, paragraph, link text, ...).

With the group component it is theoretically possible to calculate something similar to PageRank. But since most of the documents have only one incoming and outgoing link, it does not make much sense. A solution would be to introduce semantic relations between the documents. This means, that for example an attachment has a relationship to the sender of the email, publications are related through authors and references and so on. It's possible to implement that, but it's not part of this semester work. This work focus on proximity, tfidf, term positions and, of course, we have to take advantage of the iMeMex data model.

2 Single Keyword Queries

2.1 Overview

The scoring scheme uses the scores provided by Lucene to rank documents returned by a single keyword query. This score is already normalized and can be used for multiple keyword queries too (see next chapter). The Lucene score is used for name and content component without any modification. The score is modified for the tuple component to take into account the special structure of the tuple component.

The following sections describe the general implementation in Lucene and how the scoring scheme combines and modifies the Lucene score for the different iMeMex components.

2.2 Implementation in Lucene

Scoring in Lucene is provided by the abstract class *Similarity.java* in the package *search*. The abstract methods in *Similarity.java* are implemented by *DefaultSimilarity.java*. The implementation is motivated by the cosine-distance or dot-product between **document d** and **query vector q**.

The score of **query q** for **document d** is defined in terms of these methods as follows:

- **query q**
  A sequence of search terms.

- **int numTerms**
  Total number of terms in the given query.
- **int numTokens**
  Total number of tokens (words separated by whitespaces) in the given document d.
- **int matchingTerms**
  Number of terms contained in both **q** and **d**.
- **float lengthNorm()**
  Computes a normalization factor based on the total number of tokens in a document. Matches in longer documents are less precise, so implementation of this method returns smaller values when there are many tokens. The default implementation is given by $\frac{1}{numTokens^2}$.
- **float coord(q, d)**
  Computes a normalization factor based on the fraction of all query terms contained in a document. The more terms of query q are in document d, the higher the score. This considers that the presence of a large portion of the query terms in the document indicates a better match. The default implementation is given by $\frac{matchingTerms}{numTerms}$.
- **float score(q, d)**
  Assigned Lucene score for a query q and a document d[1]:

$$score_{Lucene}(q,d) = \sum_{t \in q} [tf(t) * idf(t)^2 * lengthNorm()] * coord(q,d)$$

Lucene provides this score for each document returned by the document searcher. It is called a **Hit**.

2.3 Usage in iMeMex

If the user executes a single keyword query, the query gets executed against all indexes. Every index returns all matching documents and assigns a (Lucene-) score to each document. After merging all the Hits, a document can have more than one Lucene score assigned. The scoring system has to merge and normalize the scores if necessary.

How this is done for the different indexes is explained below.

2.3.1 Group Component

The group component is not used by the query processor. Therefore, the group component does not contribute anything to ranking and scoring. No rule is required.

[1]Note that the formula used by Lucene has two additional factors: *getBoost()* and *queryNorm()*. These two factors do not affect ranking and are therefore omitted for simplicity. A detailed description is available at http://lucene.apache.org/java/docs/api/org/apache/lucene/search/Similarity.html

2.3.2 Name and Content Component

The system uses both, name and content components for finding resource views. Actually, the name component is a short descriptor of the content component. If a query does match the name of the document, we can assume that the document is in a form important for the user. Therefore, the system has to assign a higher score to documents with a matching name component.

As seen before, Lucene normalizes a score by multiplying with *lengthNorm()*. For large components with many tokens, this function returns a small value but for small components with only a few tokens a bigger one. This is exactly what we are looking for – the score for the name component gets boosted while the content component gets a lower score.

Therefore we can use the score for name and content component without any modification – Lucene does all the work for us.

2.3.3 Tuple Component

As mentioned in chapter one, the tuple component is a list of attribute value pairs. A pair contains the name of the attribute and the corresponding attribute value. There are two indexes in iMeMex for name and value. For keyword queries, the query processor has to query only the value index. The tuple name index is used for tuple queries (see chapter 4 - "Tuple Queries").

There is another big difference between the tuple and the other three components: there is not a one-to-one relationship between component and the corresponding Lucene index. A tuple component can consist of one or more attributes. Each attribute value is mapped to one index entry. Therefore if a tuple has **n** attributes, the query processor is able to return **n** times the same resource view because of **n** matching index entries for the attribute value only.

To see the problems which can arise, consider the following example with a query **q** and two resource views **RV1** and **RV2**:

query q	["imemex"]
RV1, tuple values	{attr1="imemex", attr2="imemex .. [many tokens]"}
RV2, tuple values	{attr1="imemex", attr2="imemex", attr3, attr4}

Both resource views RV1 and RV2 have two matching index entries. For both, the Lucene score for attribute *attr1* is equal to one. Resource view RV2 has a higher score for *attr2* because the length of the attribute is smaller (has less tokens). In total, RV2 has a higher score than RV1. It maybe argued that RV1 is a better match because all attributes have something in common with the query. To consider that, the total score for tuples has to be computed like this:

T = tuple component
n = $|T|$
s_i = Lucene score for attribute i

$$score_{tuple}(s) = \frac{\sum_{i=1}^{n} s_i}{n}$$

2.4 Summary

The scoring scheme has now three normalized Lucene scores in the range [0, 1]. For name and content component, the score takes into account the length and therefore the importance of the component. For the tuple component, the scoring scheme does consider how many attributes did match. Finally, the scoring scheme has two rules for single keyword queries: one for name and content component together and one for the tuple value index.

3 Multiple Keyword Queries

3.1 Overview

The rules introduced in the last chapter for single keyword queries do also apply for multiple keyword queries. With *coord(q, d)*, Lucene does consider the ratio of matching keywords and total keywords in the formula and assigns a higher score to documents containing all keywords.

Nevertheless, this has no influence on the ranking, if the query enforces all keywords, for example *["imemex **AND** scoring"]*. For this query **q** and document **d**, *coord(q, d)* is for all matching documents the same. A proximity rule applies a score based on the distance between the keywords. The smaller a distance (in tokens), the higher the score.

3.2 Influence of coord(q, d)

Beneath *lengthNorm()* which takes into account the size of the document, Lucene provides a normalization factor to score how many of the keywords are part of the document. This factor is called *coord(q, d)*, where **q** is the query and **d** a document. Of course, *coord(q, d)* has only an influence on ranking, if the query does not enforce all keywords. This is the case if no boolean operator is given. Then *["imemex scoring"]* is internally rewritten to *["imemex OR scoring"]*.

In the default implementation, coord(q, d) is the ratio of matching and total keywords. The example below with

query q = ["imemex scoring"]

shows the benefit and also the problem:

resource view	**content**	**coord(q, d)**
RV1	'imemex'	coord(q, RV1) = 1/2 = 0.5
RV2	'imemex scoring'	coord(q, RV2) = 2/2 = 1.0
RV3	'imemex ... scoring'	coord(q, RV3) = 2/2 = 1.0

RV2 and RV3 have a higher multiplier than RV1, because all keywords appear in the content of the component. This leads to a higher score. But there is no distinction between RV2 and RV3, although in RV2 the keywords are side-by-side which is normally preferred by the user. This problem is addressed by the proximity rule.

3.3 Proximity

Queries like *["imemex scoring"]* imply that the keywords should be side-by-side in the document. If this is not the case, the distance between the keywords should be minimal. In general, the smaller the average distance between the keywords, the higher the assigned score has to be.

It is possible to use this rule for all components, but it makes only sense for the name, the content and the tuple value component. The group component is, as mention before, not used for scoring and therefore not of interest. The tuple name component mostly consists of only one word, therefore it does not make sense to apply the proximity rule.

Important is, that the proximity rule calculates the score for each component independently. If the first keyword is in the name and the second in the content component, the proximity rule can not be used.

Proximity is defined upon 10 distance classes. Class 1 means that all keywords are side-by-side, class 10 means that the keywords are far away and there is no relationship between them.

Below you will find a table with all classes. The second column, $distance_{max}$, defines the maximum average distance between the keywords (in tokens). For example,

class = 5 and
$distance_{max} = 30$

means, that for class 5 the distance between the keywords has to be smaller than 30 tokens. If there are more than two keywords in the query, the average value of all distances is used (for an example see point 1 of the remarks to the algorithm).

Note that the maximum distances must be redefined for each index. For example the average length of a name component (in tokens) is smaller than of a content component. The values given below are those used for the content component:

class	$distance_{max}$	class	$distance_{max}$
1 - side by side	1	6	50
2 - near	3	7	100
3	15	8 - distant	200
4	20	9	400
5 - in the environment	30	10 - far away	600

In short, the algorithm builds now all pairs of keyword positions, calculates the average distances, maps them to a proximity vector and uses a weight vector to calculate the final proximity score. This is done for each component.

The Algorithm

1. Assumption: the query has **n** terms and $n > 1$.

   ```
   Example: query q = ["imemex AND scoring"], n = 2
   ```

2. Get the position vectors p_i of all n query terms in document d. This information is provided by the index.

   ```
   p0 = {1, 81}, p1 = {2, 82}
   ```

   In this example $\{1, 81\}$ means, that document d contains term the first term ("imemex") at positions 1 and 81.

3. Calculate the cross product called **tuples** $= p_0 \times p_1 \times \ldots p_{n-1}$ over all position vectors.

   ```
   tuples = {(1, 2), (1, 82), (81, 2), (81, 82)}
   ```

4. Calculate the distance vector **d** for each tuple in tuples.

   ```
   d = {1, 81, 79, 1}
   ```

5. Map the distance vector d with $distance_{max}$ to a proximity class.

   ```
   d[0], d[3] => class 1
   d[1], d[2] => class 7
   ```

6. Count how often a class appears and write it to the proximity vector **p**.

   ```
   p = {2, 0, 0, 0, 0, 0, 2, 0, 0, 0}
   ```

   p[0] = 2 means, that 2 keyword combinations correspond to proximity class 1: {1, 2} and {80, 81}.

7. Use a weight vector **w** and the transposed p^T of p to calculate the final score $s = p^T * w/size(d)$.

   ```
   w = {1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1}
   s = 0.7
   ```

Remarks:

- Step 4: if tuple size is greater than two (query has more than two keywords), calculate the average distance (note that the tuples values below are imaginary):

```
tuples = {(1, 2, 7), (1, 82, 180)}
  d[0] = ((2-1) + (7-2)) / 2 = 3,
  d[1] = ((82-1) + (180-82)) / 2 = 89.5
  => d = {3, 89.5}
```

- Step 7: the division through size(d) is needed to get a final score between 0 and 1.
- The algorithm must abort if less than two of the query term appear in a component.

4 Tuple Queries

Tuple queries have the form *[attributeName op value]* where

- *attributeName* is the name of a resource view attribute,
- $op \in \{=, \neq, <, >, \leqslant, \geqslant\}$ and
- *value* is either a string or a numeric value.

It is not always possible to use special scoring functions to rank the results. For example queries with the equality operator are always exact match queries. It's a match or not, there is nothing in between. Then the scoring scheme has to use other rules. The following sections explain how results from tuple queries are ranked and how the system deals with exact match queries as mentioned before.

Note that for keyword query the tuple component index is queried too. There are no operators in keyword queries, therefore rules for keyword queries and not for tuple queries are used.

It is important to realize, that there is only **one** rule to score tuple queries. This single scoring rule uses one of the functions below depending on the operator. The operator has to be saved in order to be able to choose the correct function.

Some of the functions below use the variable $long_{max}$. $long_{max}$ is equal to the Java built-in constant Long.MAX_VALUE = 9223372036854775807.

4.1 Numeric Attributes

4.1.1 "Equality" Operator

There is no selective function if the equality operator is used. Either a document is a match or not. The scoring rule can't use a special function to rank these results. Instead, general rules as *scoreLastModified* (this rule assigns a higher score to newer documents) are used.

4.1.2 "Inequality" Operator

The query *[attributeName != value]* returns all resource views RV, where the value of the attribute *attributeName* is not equal to *value*. Intuitively, the score for a given resource view RV_i has to be proportional to

$$abs(RV_i['attributeName'] - value)$$

where $RV_i['attributeName']$ returns the value of attribute *attributeName* of resource view RV_i. In other words: the bigger the difference, the bigger the score. This is realized with the following scoring function:

query = *[attributeName* $\neq$ *value]*
x = $RV_i['attributeName']$

$$score_{attr}(value) = \left| \frac{x}{value - long_{max}} - \frac{long_{max}}{value - long_{max}} \right|$$

4.1.3 "Less" & "Less than" Operator

The function to score queries like *[attributeName* $<$ *value]* or *[attributeName* $\leqslant$ *value]* is similar to the function to score inequality tuple queries. The query result contains only resource views RV, where $RV_i['attributeName']$ is smaller or equal to *value*, therefore the scoring function is much simpler. Important is, that the scoring function has to handle two cases for the 'less than' operator:

1. **query**: *[attributeName* $<$ *value]*

   $$score_{LessOp}(value) = \frac{RV_i['attributeName']}{value}$$

2. **query**: *[attributeName* $<=$ *value]*

   $$score_{LessThanOp}(value) = \begin{cases} \frac{RV_i['attributeName']}{value} & \text{, if } (value \neq 0) \\ 1 & \text{, otherwise} \end{cases}$$

4.1.4 "Greater" & "Greater than" Operator

The scoring function for queries like *[attributeName* $>$ *value]* or *[attributeName* $>=$ *value]* is apart from the missing abs() function the same as the inequality scoring function:

query = *[attributeName* $>$ *value]*
x = $RV_i['attributeName']$

$$score_{attr}(value) = \frac{x}{value - long_{max}} - \frac{long_{max}}{value - long_{max}}$$

4.2 Non Numeric Attributes

Examples for non numeric attributes are *class*, *path* or *uri*. There are no specific rules implemented to score queries like with such attributes. The scoring scheme can only rank the result set with common rules like *scoreLastModified*.

5 Structure Queries

The scoring scheme has a rule to score structure queries like *ETH/Semester*. Such queries are always executed twice: as exact match queries and as relaxed queries. The relaxed query is an expanded exact match query - every sub path is surrounded by asterisks:

query type	**query**
exact	*ETH/Semester*
relaxed	**ETH*/*Semester**

The QueryProcessor doesn't execute the whole structure query in one step. For example, for the given query *ETH/Semester*, the system executes first the queries *ETH* and **ETH**, merges both result sets and does an unnest operation to get all child nodes (this is necessary because of the *"/"* operator). This first result set is called **s1**. In a second step, the system is queried by *Semester* and **Semester**. Again the query processor has to merge these two result sets (called **s2**). The final result set **s** is the intersection of **s1** and **s2**.

It's obvious that resource views found by the original query should have a higher score than those only found by the relaxed query. These results are exactly what the user was looking for – the resource views from the relaxed query are additions and do not perfect match. For composed queries like *ETH/Semester["imemex"]* it's of course still possible that those resource views found by the relaxed query are higher ranked because the keyword query score.

In the final result set **s**, the scoring system has to keep track for each resource view how often a query relaxation was necessary:

query **q**: *ETH/Semester/work*
sub paths of q: {'ETH', 'Semester', 'work'}
result set **s**: {RV1, RV2, RV3}

resource view	**path**	**relaxations**
RV1	ETH/Semester/work	0
RV2	ETH/**Semester_7**/work	1
RV3	ETH/**Semester_7**/**Semesterwork**	2

The scoring rule for structure queries must return the full score for RV1. For RV2 and RV3 the score has to be the ratio of relaxations to number of sub

paths. This guarantees that more exact queries (exact in terms of how many sub path relaxations where necessary) have a higher score.

The above considerations result straightforward to the following formula:

$subpath_{total}$ = number of subpaths
$subpath_{relaxed}$ = number of relaxed subpaths

$$score_{structure}(subpath_{relaxed}) = \frac{subpath_{total} - subpath_{relaxed}}{subpath_{total}}$$

This rule applied to the given example above returns the following scores:

resource view	**relaxed sub paths**	$score_{structure}$
RV1	0	1
RV2	1	2/3
RV3	2	1/3

6 Query Independent Rules

6.1 Score last-modified Attribute

This rule calculates a score depending on the *lastmodified* attribute of a resource view but independent of the actual query. The *lastmodified* attribute is a field in the tuple component of a resource view.

The idea behind the rule is very simple: users are usually more interested in newer and more actual resource views. This rule should boost such resource views:

$$score_{lastmodified}(today) = -\frac{today}{lastmodified} + 1$$

Note: today and lastmodified are given in **days** since 1.1.1970

With resource views it is possible to address only parts of a file, for example a XML node. Such resource views do not have a *lastmodified* attribute. In this case, the rule has to go back in the resource view hierarchy (over the group index) to take the first *lastmodified* attribute of a parent resource view.

Now this rule applies for every query and for every type of resource view. The rule is in particular useful if the scoring scheme has no special rule to score the result set of a query. An example is the tuple query *[filesize = 147.0]*. This query returns all resource views with a file size equal to 147. It's not possible to rank such results because every resource view in the result set does perfectly satisfy the query condition (if not, the resource view wouldn't be part of the result set). With query independent rules like scoreLastModified it is still possible to rank.

7 Overall Calculation of Complex Queries

An example of an complex query is given below:

$$ETH/Semester[\text{"}imemex\text{"}]/latex[filesize > 100.0]$$

To process this user query, the query processor parses and executes the query string and returns an unordered set of resource views. The scoring scheme applies afterwards all available rules to the result set. This rules are described in the preceding chapters. Each of this rules returns a score. To be able to rank the result set, the scoring scheme calculates a total score which is equal to the sum of all individual scoring values returned by the rules.

For the example above, the scoring scheme can use the following rules (note that the scoring scheme applies **all** rules, but only the following return a score greater than zero):

- Single keyword query rule: *["imemex"]*
- Tuple query for numeric attributes rule: *[filesize > 100.0]*
- Structure query rule: *ETH/Semester* and */latex*
- scoreLastModified rule.

Per default, each rule returns a score between 0 and 1. If a rule should be more important than another, the upper bound can be changed. This is done with the method *setWeight(double w)*. According to the rule interface *ScoringRule.java*, each rule has to implement this method. The *computeScore()* method calculates in a first step how good the given resource view does match the query. This is expressed by a value between 0 and 1. Then it's multiplied with the *weight* factor. The highest possible scoring value is therefore always equal to *getWeight()*.

The query independent rule should be less important than all other rules. The rule should have only influence on the ranking if no other rule applies or if the score of two or more resource views are equal. Therefore, the *weight* has to be smaller than 1, a suitable value for example, is 0.2.

On the other hand is proximity very important for queries with more than one keyword. In particular, it is more important than the rule for single keyword queries. The *weight* factor should be higher than 1, for example 3.

Note that the *weight* factors above are only examples. The exact values have to be evaluated by tests and depend on the registered rules.

8 Implementation

This chapter explains the most important interfaces and classes of the scoring scheme.

8.1 ScoringRule.java

Package: core.org.imemex.queryprocessor.scoring

```
public interface ScoringRule {
      public double computeScore(ResourceView rv);
      public double getWeight();
      public void setWeight(double w);
      public String getRuleDesc();
}
```

This interface represents a rule. A special background have *getWeight()* and *setWeight()*. A rule has a weight assigned to control its importance. The default value is 1, but for more important rules (which should have a higher influence on ranking), it is necessary to assign a higher value. Chapter 8 explains how to use it.

- **computeScore()**
  Computes a scoring value. For non matching rules, the method returns zero. The higher the score, the better the match. The maximal possible score is equal to *getWeight()*.
- **getWeight()**
  Returns the assigned weight of a rule.
- **setWeight()**
  Sets a new weight for a rule.
- **getRuleDesc()**
  Returns a description of the rule. This is especially useful for debugging.

All scoring scheme rules have to implement this interface.

8.2 Scoring.java

Package: core.org.imemex.queryprocessor.scoring

```
public interface Scoring {
      public void registerScoringRule(ScoringRule sr);
      public ResourceView computeTotalScore(ResourceView rv);
      public int getRuleCount();
      public double getMaxScore();
}
```

This interface provides the main functionality of the scoring scheme. It provides functions to register a new rule and to compute the total score of a resource view in the result set. The functions in detail:

- **registerScoringRule()**
  Adds a new rule to the scoring scheme.

- **computeTotalScore()**
  Iterates over all registered rules. Each rule returns a score. The total sum (of all scoring values) is saved in *ScoreAttributes.java* (see next sub chapter) and later on used by a ranking operator to sort the query result.
- **getRuleCount()**
  Returns the number of registered rules.
- **getMaxScore()**
  Returns the maximal possible score. This can be used to implement a top-k algorithm.

An abstract implementation is given by *AbstractScoring.java*. *DefaultScoring.java* is the default implementation of this interface.

8.3 ScoreAttributes.java

Package: core.org.imemex.resourceviewmanager

This class contains all required scoring information. Each *ResourceView* has an instance of *ScoreAttributes.java*. The information is collected when iMeMex parses and executes the query. It contains the following information:

- A list of matching keywords.
- Information about query relaxation.
- tfidf information for each component.
- For tuple queries: the operator.
- The actual total score.

All scoring rules do use the information provided by this class.

9 Conclusions

With the presented scoring scheme and its rule, it is possible to rank the result of an iMeMex query. The scoring scheme is extendible and support simple and also complex rules. The scoring scheme provides rules for all possible query types.

Below a few points of possible improvements/extensions of the system:

- As mentioned in chapter "Introduction", web search engines like google use a ranking feature called PageRank[2] to have a query independent rule. PageRank uses incoming and outgoing links to measure the importance of a website. iMeMex does not have enough links to calculate a similar

[2]http://dbpubs.stanford.edu:8090/pub/showDoc.Fulltext?lang=en&doc=1998-8&format=pdf&compression=

value (iMeMex has only the links from the file system and they are not appropriate). An extension of the proposed scoring scheme would be to introduce semantic links between the resource views to be able to compute a score similar to PageRank.

- At the moment there is no special rule for queries containing the transitive closure operator ("//"). An improvement of the system would be to add a rule which measures the distance between the found result and the original query.
- Proximity is defined over the distance between the query terms. An improvement would be to consider the order of the query terms.

References

[1] J.-P. Dittrich and M. A. Vaz Salles. idm: a unified and versatile data model for personal information management. In *Proceedings of the 32nd VLDB Conference*, Seoul, Korea, 2006.

[2] O.R. Girard, J.-P. Dittrich, and M. A. Vaz Salles. iql: The imemex query language for personal information management. In *Proceedings of the 32nd VLDB Conference*, Seoul, Korea, 2006.