Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

Stefan Lohrum, Wolfram Schneider, Josef Willenborg

De-duplication in KOBV

gefördert von der Senatsverwaltung für Wissenschaft, Forschung und Kultur des Landes Berlin und von dem Ministerium für Wissenschaft, Forschung und Kultur des Landes Brandenburg

Preprint SC 99-05 (June 1999)

De-duplication in KOBV

Stefan Lohrum, Wolfram Schneider, Josef Willenborg

Konrad Zuse Zentrum für Informationstechnik in Berlin (ZIB)

Preprint SC 99-05

June 8, 1999

Abstract

In KOBV we offer the user an efficient tool for searching regional and worldwide accessible library catalogues (KOBV search engine). Search is performed by a distributed Z39.50 retrieval and an index based quicksearch. Due to the number of catalogues, result sets may contain a significant amount of duplicate records.
Therefore we integrate a de-duplication procedure into KOBV search engine. It is part of the distributed search and the KOBV quicksearch as well.
Main goals are the presentation of uniform retrieval results, the preservation of retrieval quality and cutting off redundant information. At least we keep an eye on efficiency. De-duplication is fully parametrizable, so that settings can be changed easily on line.

Table of Content

1. De-duplication

Part of KOBV search engine (Willenborg 1997) is a de-duplication procedure. In distributed search we perform de-duplication on search result sets, in KOBV-quicksearch de-duplication is done while creating or updating the index.

De-duplication in KOBV is performed in three steps:
1. retrieving a set of potential duplicates (search).
2. determining duplicates precisely out of this set (match).
3. joining duplicates to one record (merge).

Paramtrization of de-duplication is done by setting certain attribute weights resp. thresholds for match and merge.

1.1. Potential duplicates

Match is prepared by a function called `potentialDups`. This function has three main goals:
a) reduce the set of potential duplicate records to gain efficiency
b) standardize characters/overlook spelling mistakes
c) transform queries for specific database needs
Recall of potentialDups should be high enough so that no "real" duplicate record is missed.

Input of `potentialDups` is a (Z39.50)-query and a database name. It delivers a set of potential duplicate records for this query and database.
`PotentialDups` needs information about databases (available attributes, character normalizations, method of query transformation) so that it is able to transform the original query to an adapted query (Z39.50- or Aleph-query) which delivers the set of potential duplicate records.

Query transformation methods for delivering potential duplicates:

Method 1: get one word from title. If this word is in a stopword list take the next word etc. Then do a register scan for this word. If the result set is not too big (e.g. 100 records) get the records from this result set. If the result set is too big take another word from title and do a find for this word and so on till we have one word which has an appropriate result set.

Method 2: get the first two words and the last two words of the attribute "title" and do a register scan for these words. Take the smallest result set (word which bears much information) and do a find with attribute "title" for this word.

Method 3: take the most important attribute of the query (e.g. title or author) and do a find with this attribute for the whole value

Method 4: take the attribute "title" and do a find with this attribute for the first 15 characters

Method 5: get single words from all attributes of the query and do a register scan for these words. Then take the smallest result set and do a find with attribute "any".

Method 6: replace the original query with a query that contains a specific number of truncation symbols (ignoring spelling mistakes). For example if the original query was "publication year = 1996" with 75% accuracy it is replaced by: "publication year =?996" and "publication year =1?96" and "publication year =19?6" and "publication year =199?". A query such as "title=retrieval" with 66% accuracy is replaced with: "title=?rieval" and "title=ret?val" and "title=retrie?". 3 characters of 9 characters are truncated 3 times.

Method 7: matchcode-searching could help to gain efficiency. One method for matchcode generation could be some kind of checksum-generation, for example we generate character frequencies for author, title and publication year values. The matchcode shouldn't be so exact that spelling mistakes cannot be taken into account.

Disadvantages of Matchcode:

- rate of making mistakes (inconsistencies, etc.) is increased
- two methods (matchcode searching and distributed searching) have to be used and to be brought into line
- an extra attribute field (matchcode-field) for records has to be defined
- an extra index for the matchcode-field has to be created and maintained
- generation has to be done for all records in the database and in every updateRecord situation
- whenever the matchcode-parametrization is changed (e.g. threshold, normalization method) not only links between records have to be changed but the matchcode itself has to be changed and newly inserted into the Matchcode-Index for all records.
- matchcode-searching can only be performed in databases, where this matchcode is available, so that it cannot be used for external databases and not with Z39.50

In KOBV we use method 1.

In order to increase performance and retrieval quality we consider the frequent case that a query asks about a standard number (ISBN, ISSN, DNB-number, LoC-number, etc.). In this case no further match-step is necessary and we deliver the result directly.

1.2. Match

The match function determines, whether two records represent the same bibliographic record (duplicate records). Input of match are two records (which are potential duplicates). Match delivers true if two records are duplicate records else false. Match is based on weight functions and similarity functions for fields.
Match is efficient enough because we use it for a small set of potential duplicate candidates. Match will be accurate if it is properly parametrized.
We will test different match parametrizations. One of them will be favoured in KOBV.

First step of match is the conversion of the original record into an intermediary record format which is powerful enough to unify relevant fields of record formats. In KOBV we use MAB2 with MAB-character set. A better solution could be Dublin Core with character set ISO 8859-1 resp. Unicode.

1.2.1. Normalization

Second step of match is to select specific fields of our potential records and to normalize these fields with specific methods (see Rusch 1999).

id	**Character normalization**
1	delete Aleph-subfields
2	get the first digits
3	remove special characters which are defined in a separate set per field (e.g. point, comma, ...)
4	get all characters after a special character
5	truncate after a delimiter
6	truncate after n characters
7	capitalize/decapitalize characters
...	...

Table 1: Character normalization methods (see also Rusch 1999)

database	record fields	character normalization
KOBV	author	3, 4, 5, 7
KOBV	corporate	3, 4, 7
KOBV	title,title series	3, 4, 7
KOBV	standard number	3, 5
KOBV	publication year	2, 4
KOBV	publication place	3, 4, 6, 7
KOBV	publisher	3, 4, 6, 7
KOBV	report number	3, 7
KOBV	volume number	2, 4
...	...	...
GBV	author	3, 4, 5, 7
GBV	corporate	3, 4, 7
...	...	...
BVB	author	3, 4, 5, 7
...	...	...

Table 2: Character normalization of record fields (see also Rusch 1999)

1.2.2. Similarity of record fields

Similarity functions are widely used in information retrieval to measure vector similarities (see Salton 1983, 201ff). We measure the similarity of strings in comparing gram-representations of these strings. The comparision function then delivers the similarity of the two strings as a real number between 0 and 1. The length of the grams (2-grams, 3-grams, etc.) has an important influence on the comparison result. For short strings 2-grams seem to deliver more precise results than longer grams.

For other types of fields (e.g. date, number, ISBN, etc.) we use special field compare functions. For example two ISBN are similar either if they are exact equal or if two digits are interchanged (e.g. 0-07-054484-0 and 0-07-045484-0).

In KOBV we have two string compare functions in mind:

Field compare function 1:

We measure the *distance* of two strings by counting different patterns ("n-grams"):
An n-gram is a n-letter pattern in a string, for example "sal" is a 3-gram in "salton". An n-gram vector representation of a string is defined as:

$$A_{str,\ n} = ((a_1,\ m_1),\ (a_2,\ m_2),\ ...\ ,\ (a_p,\ m_p))$$

where a_i denotes the i'th n-gram in the string str and m_i denotes the number of times a_i appears in string str

For example $A_{salton,\ 3}$ = {(sal, 1), (alt, 1), (lto, 1), (ton, 1)}.

The string comparision algorithm builds n-gram vectors of the two input strings and subtracts one vector from the other (euclidian distance).

$$D = \sqrt{(\sum(a_i - b_i)^2)}$$

where a_i, b_i denotes the i'th element in vector A and B

We see that the longer the two compare strings are the bigger D is.

D is compared to a threshold T. If D is less than T, the two strings are declared to be similar. T can be determined experimentally. Hylton 1996 for example uses the following threshold for 3-grams in view of the length of the two strings:

```
T = 2.486 + 0.025 * Len
```

where Len is the number of elements in both vectors A and B (elements must be different!)

If we compare "salton, gerard" and "salton, gerhard" with 3-grams, then we get the distance as: $D = \sqrt{(1^2 + 1^2 + 1^2 + 1^2 + 1^2)} = 2.236$

Both strings are similar, because D is smaller than threshold T (see Hylton 1996):
T = 2.486 + 0.025 * 15 = 2.861. So D < T.

We denote *similarity* of two strings by normalizing D to values between 0 and 1. We assume our new threshold fixed to 0.8:

```
S = {4/5 + ((T - D) / (5 * T)) if D < T;
     0.8 if D = T;
     4/5 - (((D - T) * 4) / ((1 + D - T) * 5) if  D > T;
```

So in our example we get S = 4 /5 + ((2.861 - 2.236) / 14.305) = 0.8 + 0,0437 = 0,8437

We see that our algorithm produce same results for other examples:

first value	**second value**	**3-gram-difference**	**3-gram-similarity**
blue velvet	green water	4.24	0.347
blue velvet for all his clothes	a big shark swimming in green water	7.874	0.165
blue velvet	green velour	3.872	0.407
1997	1998	1.414	0.890
springer verlag	springer assoc.	3.464	0.532
springer verlag	vrlg. springer	3.606	0.486
springer verlag	springland verbund	4.123	0.388
springer verlag	verl. springer	3.0	0.734
springer verlag	springer verl.	1.732	0.878
introduction to modern information retrieval	introduction to text search algorithms in information retrieval	6.245	0.270
introduction to modern information retrieval	introduction to modern information retrieval	0.0	1.0
introduction to modern information retrieval	modern introduction to information retrieval	2.0	0.888
introduction to modern information retrieval	information retrieval	4.796	0.354
klinische psychologie teil I	klinische psychologie teil II	1.0	0.937

Table 4: Comparison of strings with compare1

Field compare function 2:

We measure the *similarity* of two strings as the ratio of equal n-grams in both strings EQ to all n-grams in the shortest of both strings: `S = EQ / LS`

Two strings are said to be similar if their compare result is above a specific threshold.

For example if we compare the two strings "springer verlag" and "spranger verlug" with 2-grams, then we get S = 10 / 14 = 0.714

We see that our algorithm produce same results for other examples:

first value	**second value**	**2-gram-similarity**
blue velvet	green water	0.0
blue velvet	green melrose	0.1
1997	1998	0.667
springer verlag	springer assoc.	0.571
springer verlag	vrlg. springer	0.615
springer verlag	springland verbund	0.643
springer verlag	verl. springer	0.769
springer verlag	springer verl.	0.923
introduction to modern information and retrieval	introduction to text search algorithms in information and retrieval	0.872
introduction in modern information and retrieval	introduction to modern information and retrieval	0.936
introduction in modern information and retrieval	modern introduction to information and retrieval	0.936
introduction in modern information and retrieval	information and retrieval	1.0
klinische psychologie teil I	klinische psychologie teil II	1.0

Table 5: Comparison of strings with compare2

1.2.3. Match functions

We use `compare` in our `match` function. We distinguish the following three `match` functions:

Match function 1: simple product measurement 1

```
boolean match1(record r1, record r2)
{ threshold = 100;
  recSimilarity = 0;
  FOR all field_i in r1 and r2 DO
  {if compare(r1.field_i.value, r2.field_i.value) = 1
     {recSimilarity = recSimilarity + field_i.weight};
  }
if recSimilarity > threshold
    return true
  else return false;
}
```

`Match1` sums up field weights of matching fields. If record similarity is larger than a specified threshold `match1` returns true else false.

We set type and weight of record fields (example configuration; example record fields):

record fields	type	weight
Author: DC AUTHOR MAB 100, ... USMARC 100, ...	String	30
Title: DC TITLE MAB 310, 331, ... USMARC 210-247, ...	String	70
ID: DC IDENTIFIER MAB 540, 542, ... USMARC 020, 022, ...	Control Number	80
Publication year: DC DATE MAB 425, ... USMARC 260c, ...	Date	30
...	...	...

Table 6: Type and weights of record fields: example configuration

Example:

We compare one new record r_N with 3 same records in our database: r_1, r_2, r_3:

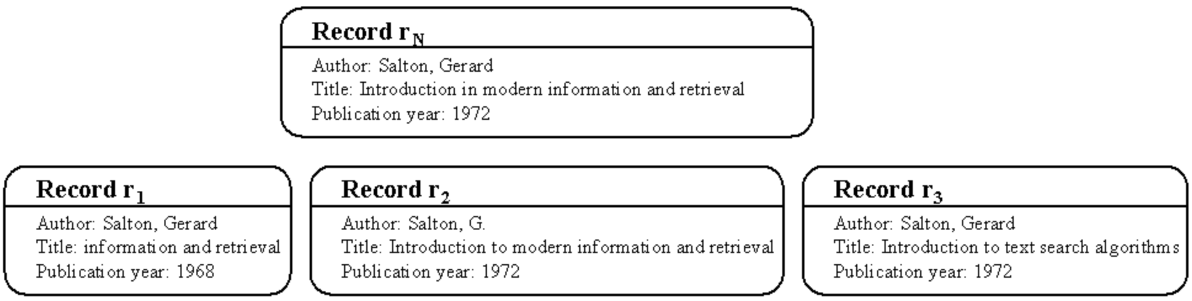

Figure 1: Records r_N, r_1, r_2, and r_3

Then our match function delivers (compare function 1; 2-grams):

```
match1(r_N, r_1) = 30 (author)
match1(r_N, r_2) = 30 (author) + 70 (title) + 30 (year) = 130
match1(r_N, r_3) = 30 (author) + 30 (year) = 60
```

If threshold = 100, r_N and r_2 are duplicates.

Match function 2: simple product measurement 2

```
boolean match2(record r_1, record r_2)
{ threshold = 0,8;
  recSimilarity = 0;
  FOR all field_i in r_1 and r_2 DO
  {fieldSimilarity = compare(r_1.field_i.value, r_2.field_i.value);
   recSimilarity =
     recSimilarity + (field_i.weight * fieldSimilarity);
```

```
  }
  recSimilarity = recSimilarity.normalizeToValuesBetween0and1;
  if recSimilarity > threshold
    return true
  else return false;
}
```

`Match2` sums up similarity comparisons of field values. Each field has a weight so that our record similarity is field specific. If record similarity is larger than a specified threshold `match2` returns true else false.

If one field of one of the two records has a value and the other does not, the compare-function delivers 100% for this field comparison.

Example:

We compare one new record r_N with 3 same records in our database: r_1, r_2, r_3 (see figure 1 and table 6):

Then `match2` delivers (compare function 1; 2-grams):

```
match2(rN, r1) =
  0,3*1,0(author) + 0,7*0,246(title) + 0,3*0,7(year) = 0,682
  normalized: 0,682/(0,3 + 0,7 + 0,3) = 0,525
match2(rN, r2) =
  0,3*0,841(author) + 0,7*0,869(title) + 0,3*1,0(year) = 1,16
  normalized: 1,16/(0,3 + 0,7 + 0,3) = 0,893
match2(rN, r3) =
  0,3*1,0(author) + 0,7*0,165(title) + 0,3*1,0(year) = 0,712
  normalized: 0,712/(0,3 + 0,7 + 0,3) = 0,548
```

If threshold = 0,8 , r_N and r_2 are duplicates.

Match function 3: positive and negative product measurement:

If we want to judge distinct positive and negative match of two records, the duplicate function looks like `match3` (see also Reichart, Mönnich 1994):

```
boolean match3(record r1, record r2)
{posRecSimilarityThreshold = 0,5;
 negRecSimilarityThreshold = 0,3;
 posRecSimilarity = 0; negRecSimilarity = 0;
 FOR all fieldi in r1 and r2 DO
 {fieldSimilarity = compare(r1.fieldi.value, r2.fieldi.value);
  if (fieldSimilarity >= fieldSimilarityThreshold)
    posRecSimilarity =
      posRecSimilarity + (fieldi.posWeight * fieldSimilarity)
  else
    negRecSimilarity =
      negRecSimilarity + (fieldi.negWeight * fieldSimilarity)}
 posRecSimilarity =
     posRecSimilarity. normalizeToValuesBetween0and1;
 negRecSimilarity =
     negRecSimilarity. normalizeToValuesBetween0and1;
 if (posRecSimilarity >= posRecSimilarityThreshold and
     negRecSimilarity <= negRecSimilarityThreshold)
          return true
 else
   return false;
}
```

`Match3` sums up similarities of fields. If this comparison is bigger than a specified threshold, the positive record similarity will be increased else the negative record similarity will be increased. Each field has a positive and a negative weight. If both positive and negative record similarities are larger than a specific threshold, `match3` returns true else false.

For each record field we set type and positive and negative weight (example configuration; example record fields; positive and negative weights from Reichart, Mönnich 1994):

record fields	**type**	**positive weight**	**negative weight**
author: MAB 100, ... USMARC 100, ...	String	20	50
corporate: MAB 200, ... USMARC 110, ...	String	20	50
title,title series: MAB 310, 331, ... USMARC 210-247, ...	String	70	70
standard number: MAB 540, 542, ... USMARC 020, 022, ...	Control Number	80	60
publication year: MAB 425, ... USMARC 260c, ...	Date	20	60
publication place: MAB 410, 415, 611, 614, ... USMARC 260a, ...	String	20	40
publisher: MAB 412, 417, 613, 616, ... USMARC 260b, ...	String	20	20
thematic number: MAB 451, 461, ... USMARC 130, 240, ...	Integer	0	70
edition: MAB 403, ... USMARC 250, ...	String	20	60
pagination: MAB 433, ... USMARC 300, ...	Integer	20	40
...	...		...

Table 7: Type, positive and negative weights of record fields: example configuration

1.3. Merge

If `match` delivers at least two duplicate records `merge` unifies them to one record. Input of merge is a set of at least two duplicate records. It delivers one merged record. Merge can be parametrized. We distinguish the following methods:

Method 1: a record quality function in which each field has a quality weight. The "quality" of the whole record is the sum of weights of these fields (which have a value). The record with the best "quality" wins. If two best records (same quality) exist, one record is selected at random.

Example:
If we have weights for author (100), title (100), year (50) and isbn (75) and the following two records r_1 and r_2:

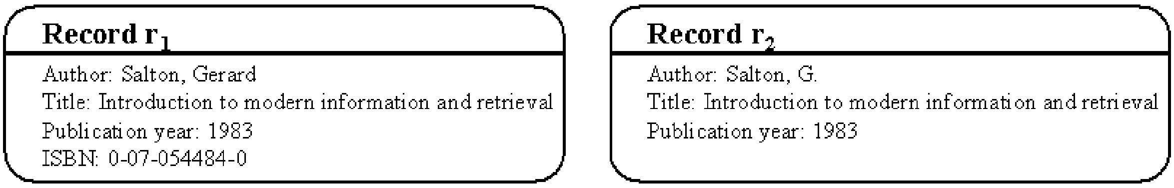

Figure 2: Quality of record r_1 and r_2

then we get for r_1 a quality of 100 + 100 + 50 + 75 = 325 and for r_2 a quality of 100 + 100 + 50 = 250, so that r_1 would be preferred.

Method 2: Fixed ranking of libraries: a) DNB; b) LoC; etc. The record of the best library wins.

Method 3: Fields are united. If one field has two values, the "best value" wins (e.g. the value with best compliance to bibliographic rules (attention: will cost performance); or the longest value, etc.). Validation checks (e.g. contradictions such as record1.year.value = 1996 and record2.year.value = 1997) can be done. The merge result is a merged record of the best fields. We produce the "best" record but possibly not homogenously.

Method 4: One record is selected at random. Not the best record wins.

In KOBV we use method 1.

2. Application of match & merge in KOBV

KOBV search engine (see Willenborg 1997) integrates major regional library catalogues by open protocols.

In KOBV we use two applications of match & merge:
1) online-match & merge for querying worldwide libraries (distributed search)
2) pre-match & merge for querying regional libraries (quicksearch)

2.1. Distributed search

In order to allow the user to extend his or her search to worldwide accessible library catalogues, KOBV-SE offers a simple and efficient application of match&merge for distributed catalogues.

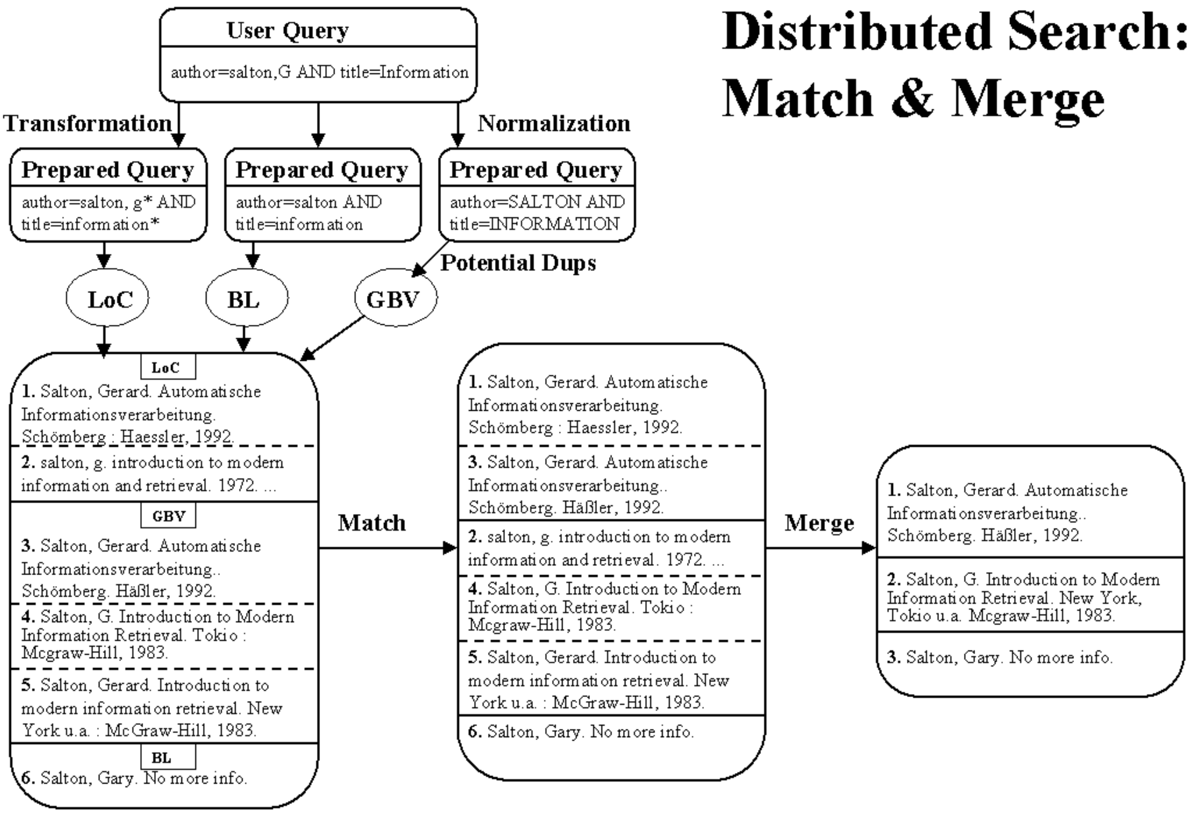

Figure 3: Distributed Search: Match & Merge

If potentialDups delivers a set of n records ($r_1, \dots, r_n$) from different library catalogues match has to be executed $n*(n-1)/2$ times:

$$
\begin{array}{lll}
\texttt{match}(r_1, r_2), & \texttt{match}(r_1, r_3), \dots, & \texttt{match}(r_1, r_n), \\
 & \texttt{match}(r_2, r_3), \dots, & \texttt{match}(r_2, r_n), \\
 & \dots & \dots \\
 & \dots & \dots \\
 & & \texttt{match}(r_{n-1}, r_n)
\end{array}
$$

We get different sets of duplicate records ($R_1, \dots, R_m$). Each of these are merged to one record so that we get merged records $mr_1, \dots, mr_m$ as our retrieval result.
One option could be sorting the result set by specific fields (pre-sorting).

If the result set of `potentialDups` is not too large, the user gets precise results with match.
If the result set of `potentialDups` is big we have to make a compromise between execution time and result quality. This could be done with parameters such as maximum execution time and maximum size of result sets. Thus an answer of the original user query can be guaranteed in reasonable time, but the result eventually does not contain all the desired information (from all desired library catalogues).

To increase performance we also think of delivering records in brief-format (Z39.50 present brief record).

Another possibility could be to present the user the result of `potentialDups` first without de-duplication. This is done with some Z39.50-browsers such as ZNavigator (ZNavigator 1998), Liman Scout (Liman Scout 1998) and Bookwhere 2000 (Bookwhere 1998) or in projects such as DBV-OSI (DBV-OSI 1998) or KvK (KvK 1998). If the user wants to de-duplicate his record sets he starts this process manually.

Advantages:

- the user can expand his search to worldwide accessible library catalogues
- new library catalogues for searching can easily be integrated
- duplicates are detected at query time
- only a small amount of data has to be stored centrally
- match & merge procedures/parametrizations can easily be changed at every time
- no inegrity costs

Disadvantages:

- dependency on library systems: library systems aren't always available.
- bandwidth: could be low (sometimes)
- response times: could be long (sometimes)

In order to realize a distributed index scan we will discuss remarks made in Search Engine Specification Requirements KOBV (see Search Engine 1998).

2.2. Quicksearch

In KOBV we offer the user an efficient method for searching regional library catalogues.
KOBV-SE prepares quicksearch in generating:

- one distributed catalogue of all regional library catalogues (KOBV-DB)
- indices for the most common regional attributes (KOBV-INDEXX).

KOBV-DB an d KOBV-INDEXX are maintained by an update-mechanism (insert, update and delete records). Update has to be done in mass production (update of many records, e.g. import of a whole library catalogue) and in every day production (update of few records, e.g. import of a one day production of one library).

KOBV-DB is a library catalogue which contains merged records of regional library catalogues. KOBV-DB does not store whole records as in central library consortia but records that have an identifier, links to full records (in other regional library systems) and a few bibliographic elements (short bibliographic record). A KOBV-DB record looks like the following one:

```
id: KOBV123
link: TUB456
link: HUB789
link: POTSDAM234
link: TUB654
link: SBB123
link: FUB678
author or creator: "Salton, Gerard"
title: "Introduction to modern information and retrieval"
date: "1972"
publisher: "Springer"
identifier:
  ISBN: 3-12-517141-5
  ISBN: 0-00-433462-0
  URL: "http://www.books.org/4711.html"
format: txt
resource type: text
```

KOBV-INDEXX is a set of indices (e.g. author, title, subject, any, author-word, title-word, ...) of regional library catalogues in Berlin and Brandenburg. Each entry of KOBV-INDEXX has an id and links to records in different library systems. An author entry for example looks like the following one:

```
id: Salton, Gerard
link: KOBV123
link: KOBV789
link: KOBV321
link: TUB456
link: HUB789
link: POTSDAM234
link: TUB654
link: SBB123
link: FUB678
link: COTTBUS987
link: TUB123
```

An index in KOBV-INDEXX is presented to the user in the following manner.

```
AUTHOR
   ...
   ...
   Salton, Gerard     (6)   [20]
   ...
   Salton, G.               [1]
   ...
   ...

TITLE
   ...
   ...
   Introduction to modern information retrieval  (3)   [10]
   Introducction to modern information retrieval       [1]
   ...
   ...
```

In entry "Salton, Gerard" the user gets information about how many records with author="Salton, Gerard" exists in KOBV-DB (the first number in brackets, here 6) and in all regional library catalogues (the second number in brackets, here 20). In entry "Salton, G." we have no record in KOBV-DB but one record in a regional library catalogue.

Updating of KOBV-DB and KOBV-INDEXX looks like:

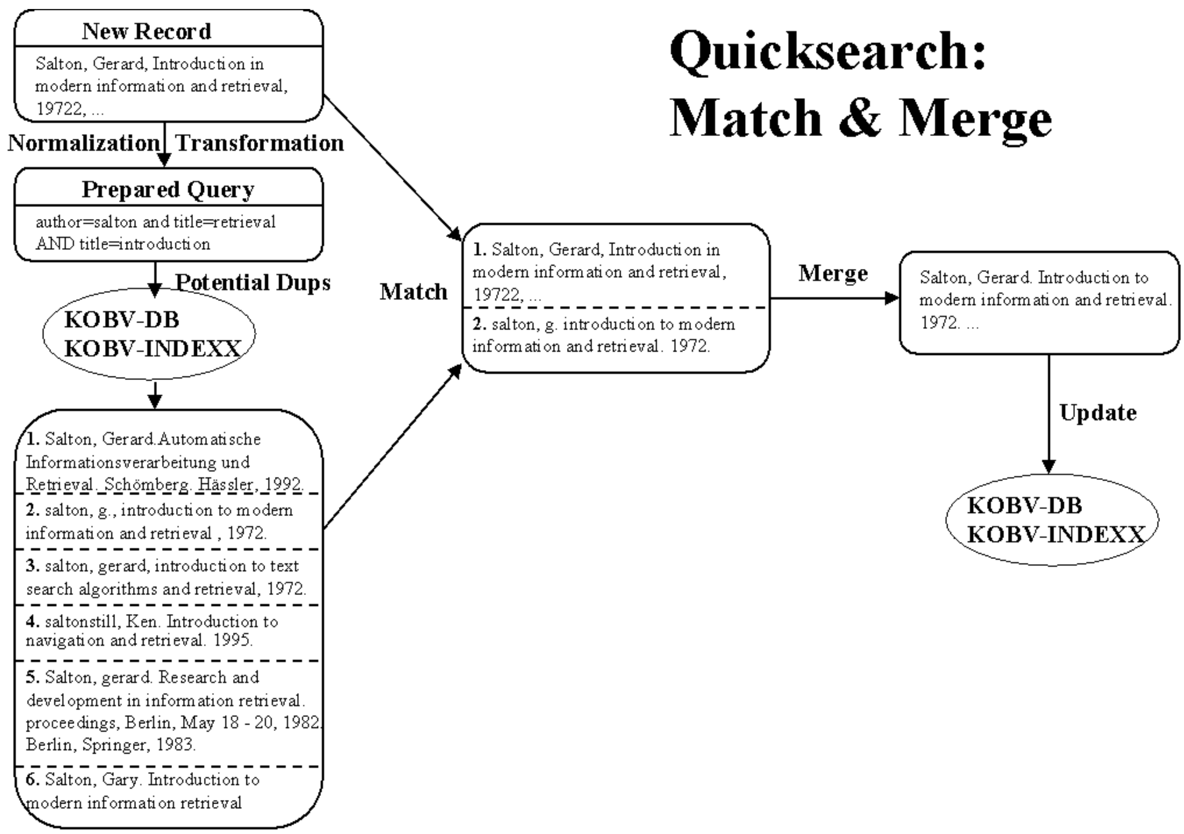

Figure 4: Quicksearch: Match & Merge

If potentialDups delivers a set of n records ($r_1, \ldots, r_n$) for a new record (r_{New}) `match` has to be executed n times: `match(`r_{New}`, `r_1`), match(`r_{New}`, `r_2`), ... , match(`r_{New}`, `r_n`).`
If no record is matching r_{New}, then r_{New} is directly updated without merging. If one record r_i is matching r_{New}, then r_i and r_{New} are merged to one record mr which is updated in KOBV-DB and KOBV-INDEXX.

The sequence of loading data into KOBV-DB can have an influence on the result. For example if we have the following 3 records:
① "Salton, Gerard. Introduction to modern information retrieval. Springer, 1972."
② "Salton, G. Introduction to modern information retrieval. 3-12-517141-5." and
③ "Salton, G. mod. informat. retriev. 3-12-517141-5. Berlin, Springer, 1972".
If we first load ① then ②, ① and ② are merged at random to ②, because the quality of ① and ② is equal. Then we load ③ and match recognizes that ② and ③ are duplicates because they have equal isbn-numbers and merge them to ③ so that we have one merged record ③ in KOBV-DB.
But if we first load ① then ③, match does not recognize them as duplicates so that we have both records ① and ③ in KOBV-DB. Then we load ② and match recognizes ① and ② as duplicates and ② and ③ as duplicates. So ① and ② are merged at random to ①. ② and ③ are merged to ③. So we have two records ① and ③ in KOBV-DB.

A solution to this dilemma could be to declare the duplicate relation as transitive . So if ① and ② are duplicates and ② and ③ are duplicates also ① and ③ are duplicates (transitive closure of our duplicate relation). So if we get more than one duplicate for one new record we have to merge them all. Then in our example we would merge ① ② and ③ to ③.

Advantages:

- if the user is searching in regional library catalogues, duplicates have not to be detected at query time.
- only a small amount of data has to be stored centrally.
- search results (small and big) without duplicates can be presented efficiently to the user.

Disadvantages:

- if match doesn't separate duplicate records from no duplicates records sharp enough, we get different database record sets (KOBV-DB) for different update sequences of records

We will test our update procedures. If this test yields to the result that it is not efficient enough we will develop a more efficient update method.
We consider the following method:
a) specify one matchcode generation method:
one method could be holding two matchcodes:

a) ISBN (for all records that have this value)
b) first two words of the author field followed by the title field followed by the year field

b) in every mass update situation (e.g. 300.000 new records) all records in KOBV-DB (e.g. 2 Mio. records) have to be exported into well specified "matchcode-files" (our two sorted files: ISBN-Index, Author-title-year-Index). In our ISBN-Index each entry has its ISBN-Value and all link ids of its belonging full records. In Author-title-year-Index each entry has its author, title and year value and all link ids of its belonginging full records.
c) we insert all new ISBN-records into our ISBN-Index (simple ISBN match).
d) for all other new records we find potential duplicates in our Author-title-year-Index. Then we perform match and merge with these records (same procedure as described in chapter 1). Finally we insert the records involved in match and merge into Author-title-year-Index.
e) We import the whole ISBN-Index and Author-title-year-Index into our library system. All indices will be rebuilt.
Disadvantages:

- while running the whole update-process no update of other records is possible.
- while loading the whole "matchcode-files", the library system cannot be used at all
- an efficient method for finding potential duplicates in file-indices has to be found and implemented

3. Parametrization of match&merge in KOBV

One of the main goals in designing match&merge in KOBV is its possibility to easiliy change its parametrization. Instead of changing the basic algorithm for every different application of our procedures we define parametrization tables to apply different methods for match&merge in a simple manner.

Parametrization of match&merge in KOBV is not simple. We will test different parametrizations with representative record sets from different KOBV-libraries.

While we build up KOBV-DB we use exactly one parametrization for potentialDups, match and merge. If we would change our parametrization while we build up KOBV-DB, KOBV-DB would become inhomogeneous.

We set the following parameters:

1. Match-intermediary-format: MAB2 with MAB-character set.

2. Potential duplicates: method 1 (see chapter 1).

3. Match-function: we try and test all de-duplication methods (match 1-3: see chapter 1.2) with following settings:

record fields	**type**	**weight**
author	String	30
corporate	String	30
title, series title	String	70
standard number	Control Number	70
publication year	Date	30
publication place	String	20
publisher	String	10

material	String	10
edition	String	40
pagination	Integer	30

Table 8: Type and weights of record fields: KOBV configuration

record fields	type	positive weight	negative weight
author	String	30	50
corporate	String	30	50
title, series title	String	70	70
standard number	Control Number	70	60
publication year	Date	30	60
publication place	String	20	30
publisher	String	20	20
material	String	20	20
edition	String	30	60
pagination	Integer	20	40

Table 9: Type, positive and negative weights of record fields: KOBV configuration

4. Merge: merge method 1 (quality function for records: see chapter 1.3) with following settings:

record fields	weight
author	100
corporate	50
title, series title	100
standard number	75
publication year	50
publisher	25

Table 10: Weights of record fields: KOBV configuration

4. Performance

4.1. Quicksearch

At this time we cannot estimate costs of our update-process for KOBV-DB and KOBV-INDEXX.

4.2. Distributed search

The following scenario is developed as a first estimation and should therefore be read cautiously. We have to verify results by tests.

Scenario 1:
We get 10 records from 10 Z39.50 Servers. If we get on average 5 records per second we get all 100 records in 2 seconds. Match has to be applied 1 + 2 + ... + 99 times (worst case). With formula n*(n-1)/2 we have 100*99/2 = 4950 match-calls. If one match(r_i, r_j) needs on average 1/10.000 sec. all comparions are done in 1/2 a second. Then we need 2.5 seconds for the whole match of 100 records.

If we have 10 Z39.50-servers the duplicate ratio will be high. We have for instance 20 merges. If one merge needs 1/10.000 sec. then all 20 merges can be done in 1/500 sec. Thus we need 2.5 sec. for the whole match&merge. If we want to sort our result set with 80 merged records we need 480 comparisons (n * log n). If one comparison is done in 1/10.000 sec sorting of 80 records can be done in 1/20 sec.
A test of Schneider 1999 showed that matching (method see Reichart; Mönnich 1994) and merging on a Sun Ultra II-Processor, 336 Mhz of 60 MAB-records needs 1/2 second based on a Perl-Interpreter-Script (without Z39.50-find&present). Optimizations are possible e.g. with C.

We see that we can mostly improve performance in decreasing the size of requested result sets and increasing the number of requested efficient library catalogues. The higher the number of requested library catalogues the higher is the merging ratio and the number of delivered records which needs bandwidth.

We can improve performance by application of the following rules (some results verified by ZACK, a Z39.50-gateway developed at the Konrad Zuse Zentrum für Informationstechnik; see Schneider 1999):

Do not perform "killer-queries": if one query delivers result sets which are too large, then we can't perform match&merge.

Minimize your intermediary record format: decrease the number of fields as much as you can.

Do not compare fields twice: if the user is searching for some attribute/fields compare has not to be done on these fields again.

Compare certain fields first: values of certain fields are usually rich in context (e.g. author, title, isbn, etc.). These fields should be compared before other fields. On average the overall number of comparisons can be highly decreased.

Hold open connections: Distributed search could work with a specific number of open Z39.50-connections to library catalogues which are important for the KOBV-consortium. With this technique we could gain half a second on average for every query (see Schneider 1999).

Use brief records: With this technique we decrease bandwidth (on average instead of 1 KB only 500 Bytes per record). Further on we decrease the number of comparisons which have to be done by match. On average we get 5 records (each 500 Bytes) per second from major Z39.50-Servers (Aleph/ZIB, Allegro-Braunschweig, Bayerischer Bibliotheksverbund, DBV-OSI; see Schneider 1999). Disadvantage of this rule is that brief record formats are not standardized yet, so that we have to be careful which fields which we use.

Sort if possible: If Z39.50-Sort is available then this technique can help to reduce the set of potential duplicates which first shall be matched&merged. But at this time Z39.50-Sort is not widely spread.

Try scan if possible: If one scan delivers 20 entries and one scan can be done in 1 second on average (see Schneider 1999) then Z39.50-scan could be an alternative to Z39.50-search.

Execute queries at low work times: If possible a technique to decrease load peaks we could execute particular queries (asyncronous) at a time when we don't have high work loads (e.g. at night).

5. Further steps

In order to increase retrieval quality we will test our de-duplication methods with representative record sets (regional and worldwide). We take samples and compare performance. If necessary we have to implement a better de-duplication method.

6. Acknowledgements

We are grateful to all who contributed their suggestions, especially to Konrad Heiming (ExLibris), Juppi Hertzmann (ExLibris), Michael Rieck (UB Potsdam) and Yohanan Spruch (ExLibris).

7. Literature/Hints

Bookwhere 1998. see: http://www.bookwhere.com/ .

DBV-OSI 1998. see http://www.ddb.de/partner/dbv-osi_ii.htm

Hylton 1996. Hylton, J. Identifying and merging related bibliographic records. Master thesis of engineering in electrical engineering and computer science, Massachusetts Institute of Technology, June 1996.

KOBV 1999. see http://www.kobv.de

KvK 1998. see http://www.ubka.uni-karlsruhe.de/kvk.html

Liman Scout 1998. see http://www.liman.com/ger/scout/ .

Reichart, Mönnich 1994. Reichart, Markus and Michael Mönnich. Dublettenkontrolle in bibliographischen Datenbanken. In Bibliothek Forschung und Praxis, Vol. 18, No. 2, 1994, pages 193-216.

Rusch 1999. Rusch, Beate. Normierungen von Zeichenfolgen als erster Schritt des Match. Berlin, Konrad-Zuse-Zentrum für Informationstechnik (ZIB), Preprint SC 99-13, June 1999.

Salton 1983. Salton, Gerard. Introduction to modern information retrieval. Mc Graw Hill, 1983.

Schneider 1999. Schneider, Wolfram. Ein verteiltes Bibliotheksinformationssystem auf Basis von Z39.50. Unpublished diploma thesis at Technische Universität Berlin, Fachbereich Informatik, Fachgebiet Wissensbasierte Systeme (WBS), 1999.

Search Engine 1998. Search Engine: Specification requirements KOBV. In: Lokales Integriertes Bibliotheksinformationssystem und Suchmaschine im KOBV (Los 1), Annex. Berlin, Konrad-Zuse-Zentrum für Informationstechnik (ZIB), June 1998.

Universe 1997. see http://www.fdgroup.co.uk/research/universe/d4-2-1.htm, Aug. 1997.

Willenborg 1997. Willenborg, Josef. Die Suchmaschine des KOBV - Spezifikationen der Anforderungen. Berlin, Konrad-Zuse-Zentrum für Informationstechnik (ZIB), TR 97-13, August 1997.

ZNavigator 1998. see http://www.sbu.ac.uk/litc/caselib/software.html.

A. Appendix: Other de-duplication efforts

A.1 Project Universe

The de-duplication process accepts a record item, compares it with all the records currently in the result set (match) and then stores it in the result set (i.e. either merges it with an item already present in the result set or adds it to the result set). This process will be called within StoreItem (see Universe 1997).

The de-duplication process is optimized by maintaining memory based indices on the attributes configured for record matching.

StoreItem (match & merge)

Before placing the record in the result set the StoreItem function will check the records in the de-duplication index. If a match is found according to the algorithm described below (match) the MergeRecord function is called, otherwise the AddItem function is called.

When finding the record which best matches the new record, user configuration of the importance of each attribute type must be taken into consideration. An example of such a configuration is given below:

Attribute Name	Attribute type	%age Match
ISBN	Control Number	95
Title	String	50...
Publication Date	Date	45
...	...	...

Table 13: Example de-duplication configuration (project Universe)

To find the best match record the steps involved would be:

Find each attribute value that matches one of the new record's attribute values. The algorithm used to find if an attribute value matches would depend on its type e.g.

String - use de-cased, unpunctuated matching
Control Number - use exact matching
Date - use date logic

For each record that has an attribute that matches find an overall percentage for all its attributes that match. The record with the highest overall match percentage is the best match.

If the best match percentage is under the match threshold (again specified by the user) then the new record is not a duplicate and will have to be added to the dedup_index table and the result set. If the best match percentage is over the match threshold then the new record is a duplicate and will have merged with both the dedup_index table and the result set.

Algorithm

```
{Set New Record Likelihood = 100%
For each record in the existing result set or until match found:
{For each data element in de-duplication profile:
{If NewRecord Data Element matches Result Set Data Element according to
rules specified by Data Element Type then:
{New Record Likelihood = New Record Likelihood
* (100 - Data Element Match Likelihood)
If New Record Likelihood > Match Threshold then
{Merge Record and update result set (MergeItem)
Notify Zserver of modified record
Finish }}}}
Add Record to Result Set (AddItem)
Notify Z Server of New Record}
```

MergeItem

MergeItem merges the new record with the matched record in the result set. Entries in the dedup_index table are also updated to take account of any new values for matching attributes.

The MergeItem function will call the ResultSet_Itemchange_Notification callback if the matched record has already been presented to the Z39.50 server.

AddItem

AddItem adds a new record to the result set and creates new entries in the dedup_index table.

A.2 KARIN

Reichart, Mönnich (Reichart, Mönnich 1994) have developed a de-duplication method based on expert system techniques (MYCIN) at the university library of Karlsruhe. Two positive weights (both fields are equal; one field of both records isn't set) and one negative weigth (both fields are not equal) were used to compute an overall weight function which sums evidences.

A.3 Liman Scout

Liman Scout (Liman Scout 1998) is a Z39.50-Client, which supports record-formats USMARC, UNIMARC and SUTRS.

The user selects match attributes (author, title and date) and a match-accuracy between 0 and 100%. Liman Scout then computes the duplicate-function with these settings.

A.4 ZNavigator

ZNavigator (ZNavigator 1998) is a Z39.50-Client, which supports record-formats USMARC, UNIMARC, UKMARC, SUTRS and GRS-1.

The user selects match attributes (author, title, edition, publication and originator). He chooses how many characters from the beginning of that field he wants to compare.