RESEARCH REPORT SERIES
(Computing #2007-1)

BigMatch:
A Program for Extracting
Probable Matches from a Large File

William Yancey

Statistical Research Division
U.S. Census Bureau
Washington, DC 20233

Report Issued: June 15, 2007

Statistical Research Division
U.S. Bureau of the Census
Washington, D.C. 20233

Research Report Series

BigMatch: A Program for Extracting Probable Matches from a Large File

Revised: June 25, 2007

William E. Yancey

BigMatch: A Program for Extracting Probable Matches from a Large File

Revised: April 24, 2007

Abstract

We present documentation for running the **BigMatch** program, a new record linkage tool for use in matching a very large file against a moderate size file. For each of several blocking criteria, the program can extract likely matching record candidates from the large file without requiring sorting of either file.

Key words: record linkage, software

Background

The **BigMatch** program is based on the Census Bureau record linkage program by Winkler *et al.* For this and other record linkage programs, record pairs from two files are brought together to be compared when they agree on a specified blocking criterion. In order for the best matching pairs to be found, record linkage projects generally require successive passes with the application of several blocking criteria, each of which requiring its own file sorts. When one of the files is very large, the time required to sort the file can become prohibitive.

The **BigMatch** program allows one to run several different blocking criteria for every large file and a moderate size file requiring the large file to be read just once and without requiring prior sorting of either file. Required sorting is done on a key list in memory, and the time required to run the program is generally modest. For each blocking criterion, the program outputs a file of records from the large file that are plausible matches to records in the moderate file.

The standard Census Bureau record linkage program features one-to-one matching which results in each record being paired with its most likely match within its blocking group. The **BigMatch** program does not do this, so that an output file may contain several records from the large file that were stored as likely matches to the same record in the moderate file. The purpose of the **BigMatch** program is to function as a preprocessor that can efficiently extract

smaller files from a very large file so that these smaller files can be used efficiently with standard record linkage software. However, the **BigMatch** program has also been used for deduplicating a single file. If a file contains multiple duplicates, then the absence of one-to-one matching can be advantageous, provided that the file can fit in core memory.

Further information about the record linkage software can be obtained from william.e.yancey@census.gov or william.e.winkler@census.gov.

Extracting Probable Matches from a Large File for Record Linkage

Program Overview

The purpose of the **BigMatch** program is to extract subfiles of plausible match records from a very large file by making only one sequential pass through the very large file. The large file never has to be sorted. Suppose that we want to find matching records in a very large master file, which we will designate as the *Record* file, that correspond to records in a moderately sized file, which we will designate as the *Memory* file. We define a file to be of moderate size if it can comfortably fit into core memory. The **BigMatch** program allows the user to specify several blocking criteria, each with its own matching field parameters. The program reads in the *Memory* file and creates a table of the keys for each blocking criterion. The program then proceeds to read in records from the large *Record* file. For each blocking criterion, the program determines whether there are any records from the *Memory* file which have keys that match the key of the *Record* file record. For all *Memory* file records with matching key value, a matching comparison weight is computed with the *Record* file record. If any of the comparison weights exceeds a cutoff value, the *Record* file record is written out to a subfile corresponding to that blocking criterion. Another file contains the pairs matching field values.

Program Changes for This Revision

The main change in the program is to allow more than one *Record* file to be compared to the same *Memory* file. Since there may be several files being compared, we have changed the names of the output files to reflect the names of the input files being compared. Old hands may notice that the roles of the previously designated *A* and *B* files have been switched, so we have adopted the names *Memory* file and *Record* file to try to minimize confusion.

Running the Program

Compiling the Program

The program `bigmatch.c` can be compiled using a C compiler. On a UNIX machine, one can use the command

```
cc –o bigmatch bigmatch.c –lm
```

where the –lm flag instructs the compiler to link with the C math function library. This flag may be unnecessary if the C math function library is in the

linker's search path. One may also use optimization flags to enhance performance. The program **BigMatch** currently compiles and runs on Compaq Alphas, Sun workstations, and Windows PCs.

The software should compile and run correctly on almost all machines with C compilers. The software is fast (~300,000 pairs per second on an Itanium computer when using 10 blocking criteria. A million record file (10^6) will need approximately 1.5 gigabytes of RAM if 10 blocking criteria are applied. The memory-data structures contain all the strings that are used for blocking and matching. The memory-data structures contain a substantial number of pointers for rapid retrieval and comparison of pairs of records.

Program Input

The program reads in two input files:

- parmn.txt
- parmf.txt

The file `parmn.txt` simply contains the paths to the input files. On the first line is the path to the *Memory* file and on the subsequent lines are the paths to the *Record* files. If the program is being used for deduplication of a file, the same file path is used on both lines.

The file `parmf.txt` contains the parameter information that controls the program execution. Refer to the sample parameter file at the end of the document.

The first line contains 9 integers:

1. The number of blocking criteria or blocking strategies
2. The number of sequence fields
3. The output cutoff flag
4. The number of ID fields
5. The print output flag
6. The record duplicate flag
7. The number of *Memory* file records
8. The length of an *Record* file record
9. The length of a *Memory* file record

The program allows the user to test the *Record* file against several blocking strategies simultaneously. The first number tells the program how many blocking strategies are to be used. The current maximum number of blocking strategies is 100. The sample parameter file indicates 6 blocking passes.

The sequence fields are the fields that contain the unique record identification number for each record in its respective file. Here the user indicates the number of fields that make up the sequence number. The sample file indicates 1 sequence field.

A positive output cutoff flag indicates that the user intends to supply cutoff values for the matching weights for the *Record* file subfile output. A zero indicates that the program should use the default values, which are currently 16 and 0. In the sample, the output cutoff flag is set to true.

The ID fields refer to a unique entity identifier, such as the Census identification number, which should be consistent across all files. This ID field is here for program analytic testing purposes and does not play a role in the ge n-

eral matching. If no ID number is to be used, this field should be set to 0, as in the sample file.

The print output flag is set to 1 if the user wishes to supply cutoff values for the matching weights of the pairs of matched *Memory* file and *Record* file records to be printed out for analytic purposes. If this flag is set to 0, the default cutoff values of 12 and 0 will be used. In the sample, the print output flag is set to true.

The record duplicate flag is set to 1 if the user is trying to find duplicate records within a file. When this flag is set, the program only computes comparison weights when the *Memory* file record sequence number is less than the *Record* file record sequence number. This prevents computing the matching weight of a record with itself or with computing matching weights for both symmetric pairs of records. Hence if a file contains n duplicated records, instead of the program outputting n^2 pairs, it outputs $n(n-1)/2$ pairs. When not searching for duplicates within the same file, this flag should be set to 0. The sample file sets the duplicate flag to 1, indicating that this is a deduplication run.

If the number of records in the *Memory* file is known, it can be entered here. The number of *Memory* file records can be obtained from the report of the **blokstats** program, as described below. If the number of *Memory* records is not known, this input integer should be set to 0 and the program will count the number of *Memory* file records, as is the case in the sample file.

The *Memory* and *Record* files are assumed to be flat ACCII files of fixed record length. In these last two positions, the user informs the program of the length of the *Memory* file record and the *Record* file record respectively. The sample parameter file sets the (maximum) record length for the files to 100 characters long.

The second line of the `parmf.txt` file is a row of integers providing the number of blocking fields for each blocking strategy. The number of integers on this line should equal the first number on the first line of the file. The sample parameter file shows that the 6 blocking strategies have 6, 6, 2, 3, 3, 4 blocking variables respectively.

The third line of the `parmf.txt` file is a row of integers providing the number of matching fields for each blocking strategy. Again, the number of integers on this line should equal the first number on the first line of the file. The sample file shows that all blocking passes have 6 matching fields. In general, the number of matching fields does not have to be the same for every blocking run.

The next lines of the `parmf.txt` file provide the parameters for the blocking fields and matching fields for each blocking strategy. First are listed the blocking field parameters and then the matching field parameters. The number of lines for the first set of blocking fields should equal the first integer from the second line of `parmf.txt`. Only records with agreeing blocking keys will be compared. Each line contains:

1. A blocking field name (up to 20 characters)
2. The start position in the *Memory* file of the blocking field

3. The length of the blocking field
4. The start position in the *Record* file of the blocking field
5. The length of the blocking field
6. The blank filter flag

The blank filter flag allows the user to specify how the program should deal with record pairs with blank key components. If a blank filter flag is set to 1, then if the records have this blocking key component blank, then the comparison weight for these records is not computed. If the blank filter flag is set to 0, then if the records have this blocking key component blank, the records are considered to agree on this key component. Hence if any key components with blank filter flag set to 1 are blank, then the record weight comparisons will not be made. A key component is considered to be blank if it consists of all space characters or all '0' characters.

In the sample file, the next 6 lines show the first set of blocking variables. Note that the field lengths for the *Memory* file and *Record* file blocking variables must be the same. All of the blank filter flags are set to 1 so that any *Record* file record that has any of its blocking field component values blank will be skipped.

Note that if there are more than one *Record* files, they all must have the same layout.

The next lines of the `parmf.txt` file provide the parameters for the matching field parameters. The number of lines of matching field parameters should equal the first integer on the third line of the file. These are the fields in the file records that are compared to compute a matching weight. Each line contains:

1. A matching field name (up to 20 characters)
2. The start position in the *Memory* file of the matching field
3. The length of the matching field
4. The start position in the *Record* file of the matching field
5. The length of the matching field
6. A null flag (non-operational)
7. The field comparison type
8. The conditional matching agreement probability
9. The conditional non-matching agreement probability

For the field comparison type, one should choose among the following options:

- **c**—exact string comparison
- **uo**—string comparator allows for some typographical variation
- **p**—numerical comparison for age
- **y**—numerical comparison for year

In addition, there is a modification that indicates checking for field inversion. The program will test a field for inversion with the next matching field if the field comparison type has an '**i**' appended to it. The last option is for exploring whether a pair of matching fields has its values switched, *e.g.* first and last names reversed. The field with comparison type with '**i**' appended

will be considered as well as the cross values with the next field in the matching field variable list. The comparison type for this next field will be used for the agreement weight calculations. The field inversion comparison option can be used only once in a matching fields list. Note that the inclusion of this option for a pair of string value matching fields can substantially increase the program running time.

In addition, there is an '**s**' comparison for numeric street names of the form "1st" and "143rd." Preprocessing is needed to put street names in consistent forms (*i.e.* convert "First" to "1st"). The '**s**' comparison is used for address matching.

The sample file shows 6 lines specifying the matching variables. Again, the field lengths for the *Memory* and *Record* files should agree. Note that for the second blocking run, the comparison type for "last" (last name) is **uoi**, indicating that it should be compared using the string comparator and that both (last, last), (first, first) but also (last, first), (first, last) should be considered for comparison values for these first two matching fields.

When using **BigMatch** for extracting subsets of plausible matches from a large file, the agreement probabilities can be rough general estimates. The first number represents the conditional probability that the two records agree on the matching field value given that the two records represent a match. The second number represents the conditional probability that the two records agree on the matching field value given that the two records do not represent a match. The agreement weight for this field value for two records is based upon the ratio of these two numbers. A larger ratio implies a stronger distinguishing power for that matching field. Presumably the ratio should always be larger than 1.

When using **BigMatch** for deduplication of a file, one is trying to identify specific duplicate pairs, so more precise probability estimates may be helpful. In the full record linkage program, these conditional probabilities are computed from maximum likelihood methods based upon the distribution of the agreement patterns in the set of pairs of records. At this time, the programs for computing agreement pattern counts require sorting the file by blocking key values.

After the matching field lines have been completed, if the output cutoff flag has been set to 1 on the first line of the file, then the next line contains a pair of numbers representing the comparison weight cutoff values for the output files. For the output cutoff values, actually only the lower cutoff value is really used, since the program outputs an *Record* file record as a plausible match candidate if it finds a *Memory* file record with an agreeing key which has a matching weight above the lower output cutoff value. Thus the size of the matching candidate output files can be adjusted by varying the lower output cutoff.

If the print cutoff flag has been set to 1, then the next line contains a pair of numbers representing the upper and lower cutoff values for the matching pair output files. If a record from the *Record* file has a key value that agrees with the key of a *Memory* file record, the matching weight of the record pair is

computed. If the matching weight is between the upper and lower print cutoff values and between the upper and lower output cutoff values, then the matching information of the two records is printed to a print output file. When using **BigMatch** to extract plausible matching candidates from the *Record* file, this output can be useful for analyzing how the program is performing its matching comparison computations. By specifying values of upper and lower print cutoff within the bounds of the output cutoffs, one can reduce the amount of output and concentrate one's analysis on a smaller range of the matching weight comparison function. On the other hand, if the program is being used for deduplication, these matching pair files are probably the main output of interest, and the two pairs of output cutoffs should probably have the same values.

In the sample, the first two blocking runs have both output and print cutoff values set to 100 and 0, while the last four blocking runs have the lower cutoff values raised to 4.7.

The listing of blocking fields, matching fields, and cutoff values repeats for each blocking strategy. As before, the numbers of blocking fields and matching fields must agree with the numbers indicated in the second and third lines of the file. These listing are repeated for the number of blocking runs indicated on the first line of the file.

After all of the blocking and matching field sets have been completed, the sequence fields are listed. The number of sequence fields must agree with the number indicated in the second integer of the first line of the file. The total sequence field should be a unique file record identifier. This is necessary when using the program for deduplication of a single file. Each line should contain:

1. The sequence field name
2. The start position of the sequence field in the *Memory* file
3. The length of the sequence field
4. The start position of the sequence field in the *Record* file
5. The length of the sequence field

The sample file has a single sequence field that starts at the first character of the record and is 40 characters long. The field lengths for both *Memory* and *Record* files should be the same.

Program Output

There are three types of output files from the **Bigmatch** program. *Record* file subfiles

- Record pair matching field information
- Summary data

The output file names are created from the *Memory* and *Record* file names. If the file `parmn.txt` lists the *Memory* file and *Record* file names as *path*`/memfile.*` and *path*`/memfile.*` respectively, then the output data files will have prefix `memfile-recfile.` In order for the program to parse these names, it looks for the last slash '/' in the file path. If the program is running on a system that uses a backslash '\' instead for its file path delimiter, then one needs to use the appropriate preprocessor definition at the

beginning of the program file. If file names without paths are used, then this is irrelevant.

For the *n*th blocking strategy, the subfiles `memfile-recfile_Sub`*n*`.dat` contain complete *Record* file records that have a key that matches a key of a record in the *Memory* file where the record pair has a matching weight above the lower output cutoff value.

For the *n*th blocking strategy, the program prints out matching information for each record pair where the keys agree and the matching weight is between the lower and upper print cutoff values (and the lower and upper output cutoff values). The program prints out the matching weight, *Memory* file sequence number, *Record* file sequence number, common key value, *Memory* file record matching field values, and *Record* file matching field values. Any such record pairs get printed out to `memfile-recfile_Pairs`*n*`.dat.`

The file `summ.dat` contains summary information about the program run for diagnostic purposes. The file gives the total number of *Memory* file and *Record* file records read. For each blocking strategy, it gives the number of distinct *Memory* file keys and the number of *Record* file records output to the subfile. Then there is a table which lists each *Memory* file key, the number of *Memory* file records with that key, and the number of *Record* file records with that key which were output to the subfile.

Note that **BigMatch** keeps track of the *Memory* file records that have been compared to a given *Record* file record, so that it does not recompute matching weights for the same records pairs for later blocking strategies. For example, if the sequence of blocking strategies is chosen to be initially most restrictive and subsequently less restrictive for each blocking run, then the subsequent output files will contain only new records found from the widening search and will not contain records printed out for previous blocking runs.

Reformatting Program Output

The output to the files `memfile-recfile_Pairs`*n*`.dat` is not very analyst-friendly. The records are in the order that they were computed and the output is not easily readable. The matching data, the *Memory* file match field values, and the *Record* field values are separated by a tilde character '~'. If the files are on a UNIX system, there are two scripts that can format the output in a more usable form. The script **detil** removes the tildes and replaces them with a new line and the script **printpairs** sorts the records by decreasing match weight and pipes the result to **detil**. The resulting output file is called `pairs.out` and it

- Is sorted by decreasing matching weight
- Has each set of record pair information printed on 3 lines
    - First line has matching weight, sequence numbers, and key value
    - Second line has *A* record matching field values
    - Third line has *B* record matching field values

These programs are used with the full matching program to aid the user in determining cutoff values for designated links and non-links. We should note that unlike the full matching program, the output of the **BigMatch** program does not necessarily have to be one-to-one. That is, the output file of record pairs may have *Memory* file records paired with more than one *Record* file record and *Record* file records paired with more than one *Memory* file record. In order to use the programs you need to:

1. Use *chmod* to make the scripts **detil** and **printpairs** executable
2. Enter **printpairs** *filename* on the command line
3. If several such files are to be reformatted, rename the output file `pairs.out` to something that indicated the source of the original file.

Evaluating Blocking Strategies

The time that the **BigMatch** program takes to run is most affected by the number of *Memory* file records that have the same key. Each time an *Record* file record is found with a given key, the matching weight is computed with every *Memory* file record that has this key. A particularly inefficient case can occur when a blocking strategy produces a large number of *Memory* file key records with a blank key. Every time a *Record* file record is found with a blank key, it will be compared with all of these *Memory* file key records even though there is not much evidence of the records being similar. As noted above, if it is known that this is happening, it can be avoided by using the blocking null flags, as noted above.

As a tool to help the user formulate more efficient blocking strategies, one can use the program **blokstats**. The **blokstats** program reads in only the *Memory* file and the parameters for the *Memory* file blocking strategies. It prints out a report file that indicates the number of *Memory* file records per key value and the number of blank keys. The user can use this information to adjust the blocking strategies or the blocking null flags before reading the large *Record* file.

The program `blokstats.c` can be compiled with a C compiler. On the UNIX machine, there is no need to link in the math library. As with the **BigMatch** program, there are two input parameter files. One file is `parmn1.txt`, which contains one line that gives a path to the smaller *Memory* file. The second input parameter file is `parmb.txt`, which gives a description of the *Memory* file layouts. The first line has a sequence of 3 integers that provide the following information:.

1. The number of sets of blocking criteria or blocking strategies—There is not a hard-coded upper limit for the number, but more blocking strategies require more memory
2. The number *n* of desired most common Memory file keys to be reported out
3. The total length of a record in the *Memory* file

The next line contains the number of blocking fields for each blocking strategy, so that the number of entries on this line equalsw the first number in

the line above. The sum of the numbers on this line equals the number of blocking fields listed on the lines below. The next lines list the blocking fields. They have the blocking field name, the starting position of the field in the *Memory* record, and the length of the blocking field.

The output file is `blkrpt.dat`. It lists the total number of *Memory* file records read, then for each blocking strategy it lists the number of distinct keys, the average number of *Memory* file records per key, the number of records with blank keys (if any), and the *n* keys with the largest number of records, where *n* is the number specified in the parameter file above. A blank key is defined to be one that consists entirely of space characters or entirely of '0' characters.

Sample Parameter File

```
6 1 1 0 1 1 0 100 100
6 6 2 3 3 4
6 6 6 6 6 6
st                    5 2    5  2 1
cou                   7 3    7  3 1
tract                10 6   10  6 1
block                16 6   16  6 1
nf_init              60 1   60  1 1
l_init               44 1   44  1 1
last                 44 15  44 15 0 uo  0.90 0.10
first                60 13  60 13 0 uo  0.90 0.10
middle               74  1  74  1 0 c   0.70 0.30
age                  76  3  76  3 0 p   0.80 0.20
mob                  80  2  80  2 0 c   0.80 0.20
dob                  83  2  83  2 0 c   0.80 0.20
100 0
100 0
st                    5 2    5  2 1
cou                   7 3    7  3 1
tract                10 3   10  3 1
```

```
block                16 6   16  6 1
mob                  80 2   80  2 1
dob                  83 2   83  2 1
last                 44 15  44 15 0 uoi 0.90 0.10
first                60 13  60 13 0 uo  0.90 0.10
middle                74  1  74  1 0 c   0.70 0.30
age                  76  3  76  3 0 p   0.80 0.20
mob                  80  2  80  2 0 c   0.80 0.20
dob                  83  2  83  2 0 c   0.80 0.20
100 0
100 0
fname                60 13  60 13 1
lname                44 15  44 15 1
last                 44 15  44 15 0 uoi 0.90 0.10
first                60 13  60 13 0 uo  0.90 0.10
middle                74  1  74  1 0 c   0.70 0.30
age                  76  3  76  3 0 p   0.80 0.20
mob                  80  2  80  2 0 c   0.80 0.20
dob                  83  2  83  2 0 c   0.80 0.20
100 4.7
100 4.7
fname                60 13  60 13 1
linit                44  1  44  1 1
age                  76  2  76  2 1
last                 44 15  44 15 0 uoi 0.90 0.10
first                60 13  60 13 0 uo  0.90 0.10
middle                 74  1  74  1 0 c   0.70 0.30
age                  76  3  76  3 0 p   0.80 0.20
```

```
mob                  80  2  80  2 0 c   0.80 0.20
dob                  83  2  83  2 0 c   0.80 0.20
100 4.7
100 4.7
finit                60  1  60  1 1
last                 44 15  44 15 1
age                  76  2  76  2 1
last                 44 15  44 15 0 uoi 0.90 0.10
first                60 13  60 13 0 uo  0.90 0.10
middle               74  1  74  1 0 c   0.70 0.30
age                  76  3  76  3 0 p   0.80 0.20
mob                  80  2  80  2 0 c   0.80 0.20
dob                  83  2  83  2 0 c   0.80 0.20
100 4.7
100 4.7
finit                60  1 60  1 1
linit                44  1 44  1 1
mob                  80  2 80  2 1
dob                  83  2 83  2 1
last                 44 15 44 15 0 uoi 0.90 0.10
first                60 13 60 13 0 uo  0.90 0.10
middle               74  1 74  1 0 c   0.70 0.30
age                  76  3 76  3 0 p   0.80 0.20
mob                  80  2 80  2 0 c   0.80 0.20
dob                  83  2 83  2 0 c   0.80 0.20
100 4.7
100 4.7
seq                  1 40 1 40
```