**Privacy Preserving Record Linkage: Linkja**

**Overview**

Linkja is designed to uniformly de-identify patients and securely link them across sites e.g., health systems, public health records, criminal justice system etc. without exposing underlying PHI. It is open source and available under General Public License. There are 3 main modules:

1. **Salt engine and project, user and key management interface**

The salt engine generates encrypted shareable salt files. The engine can be used as a stand-alone utility or coupled with a web interface. The web interface enables project managers to create projects, add users and authenticate users. Authenticated users upload their RSA public keys[[1]](#footnote-1) and download the asymmetric encrypted salt file.

Language: Java

1. **De-identification: Data standardization, exclusion, and hashing**

This module includes a data pipeline to digest and validate data, standardize data, manage data exclusions, create composite identifiers from patient identifiers, and hash them using SHA512[[2]](#footnote-2) algorithm and the shared Salt.

Language: Java, SQL-batch scripts

1. **Disambiguation**

This module allows the aggregator to merge files, disambiguate the hashes and assign master (global) patient ID to matched and non-matched patients using deterministic algorithms.

Language: Java (with SQLite)

Each module is available as a separate tool in multiple formats and can be used independently as long as the inputs for each module meet the general requirements.

**Module 1: Salt engine and project, user and key management interface**

The site responsible for generating and distributing the salt also termed as Key Master, should be either a participating site or any other site not aggregating the hashed data generated from the shared project salt. Each participating site that should receive the shared project salt, generates asymmetric public-private RSA[[3]](#footnote-3) 2048-bit key pair, sends the public key to the Key Master and receives the asymmetric encrypted salt file containing the shared salt. The shared project salt is same across all sites for a given project (or quarter or any agreed time period).

Currently, this module is available as a desktop application that can be downloaded from Github. In future, we intend to provide a web interface that can be hosted by the Key Master. The web interface will provide additional project and user management functions. Below are some features/functionalities:

1. The Project PI (or PI designated person) can create a new project and invite the users from participating site to join.
2. The invited users are authenticated and will be able to upload their public key and download the encrypted Salt file

There are 4 main stages:

**1. Project creation and management**

During this stage, the PI and/or participating sites should decide whether they will use web interface or command line tool for generating salts. Each site should be assigned a unique site id and site name. The Key Master role should be assigned or selected. Protocol to share the public key (either via email, bitbucket etc.) and encrypted salt file should be agreed upon (sFTP, secured email etc.).

Note: The role of Key Master can be any site other than the site aggregating data for that project and the aggregating site should not receive the salt used to hash the data they are aggregating, unless this is explicitly intended.

**2. Site-user management**

During this stage, the authorized user from each site should generate public-private RSA key pair. The site-user should save the private key in a secured location as this will be an input for the hashing module. The public key should be sent as a text file to the Key Master based on agreed protocol.

**3. Salt generation**

The Key Master will use the salt engine (Appendix A) to generate asymmetric encrypted salt files for each participating site (eligible to receive salt to generate hashes) from the inputs received from above 2 components.

**4. Salt distribution**

The Key Master will distribute the encrypted salt file to each participating user based on the protocol agreed in #1. The salt file is specific for each site and can be opened with site’s private key.

**Module 2: De-identification: Data standardization, exclusion, and hashing**

The de-identification module generates crosswalk between sites’ patient id and derived patient id hash along with composite patient identifier hashes. All sites participating in the project that want to de-identify and link their data should use this module with the encrypted Shared salt file received from Module 1.

Due to varied nature and quality of input data sets, standardization of data is critical. The design behind this module is to help sites at all levels of data processing sophistication and provide enough flexibility to pre-process and review their data. This module is available as a stand-alone (no web service needed) Java application and SQL-windows batch scripts. Although each program has different implementation, the data normalization rules are same, and hash outputs will be same for same input data from both programs.

In Java implementation, the data goes through a processing pipeline: ingestion, validation, normalization, exclusion checking, permutation, and hashing. Data pipeline components (Image 2) are explained in more detail below:

**1. Data ingestion**

In this stage, the data are read from the source and prepared for processing. There are 3 main inputs: source data, encrypted salt file, and random date. Please review data dictionary (Appendix C).

**2. Data validation**

There two types of validation checks:

1. Process stops: If these requirements are not met, processing stops until they are fixed. All the required fields should be present in the source data (ID, first name, last name, and date of birth), the ID should be unique (e.g. use system level identifier instead of facility level[[4]](#footnote-4)) for each record, the salt value should be 13 or more characters, and data format should meet tool specific requirements (Appendix B). The encrypted salt file should meet the output file requirements (Appendix A).
2. Warning: If there are null or blank values in the required data fields or the length of name is less than 2 characters, the processing continues, valid data are processed and the invalid records are not processed. The invalid data can be reviewed, fixed and re-processed later.

**3. Data normalization**

Sites might have different data standards and data formats. To ensure data standards are same, the data are normalized across sites. The tool goes through several normalization routines: remove suffixes & prefixes, remove extraneous characters, capitalize first and last name, truncate SSN to last 4 numbers, set SSN to blank if truncated length < 4 or 0000, and cast date as YYYY-MM-DD.

In our original design, all repeating SSN series (e.g., 9999,1111,2222 etc.) were set to blank. It is recommended that this be included in next build. The suffixes and prefixes can be edited in the Java config file and in the SQL functions script.

**4. Exception checking**

Patient records containing generic names (e.g. baby girl, unknown) when matched, create an unreal world situation. Example two new born babies born on same day and named baby boy should not be assigned same global id since it is not possible that they are same person. However, it is important that these patients be part of the cohort since an encounter did occur and could contain pertinent information (e.g. Patient 0 in network-wide measles outbreak). In current build, the sites bear the burden of pre-processing the data and setting this exclusion flag on their end.

In next build, users will have flexibility to set the exclusion flag from the tool itself, instead of having to pre-process their data. If the exclusion flag is set to 1 on a record, then during disambiguation, it will not be matched and patient will be assigned unique global ID.

**5. Permutations**

The data are permuted two ways:

1. Name derivatives: Hyphen (or space) separated last names derive two last names (e.g. Smith-Garcia are derived as Smith and Garcia) and are hashed as additional records with all the other identifier values remaining constant. The derived names are flagged and during field combination, only full name combinations are allowed (partial name combinations are not allowed).
2. Field combinations: The patient id composite is combination of patient id and difference between private date (user provides this value) and patient date of birth. 10 composite identifiers are generated from patient first name, last name, date of birth and last 4 SSN (Appendix C).

**6. Hashing**

Permuted data are hashed using SHA512 algorithm[[5]](#footnote-5). Patient id composite is hashed using the private salt and all other composite identifiers are hashed using the shared salt. For hashes that require SSN, if there is no SSN, then they will remain blank or null.

SQL-batch scripts perform above data validation, normalization, permutations and hashing steps and generate same hash values. Hashing module makes a distinction between data excluded from hashing (i.e., invalid data) and data excluded from matching (i.e., records with exclusion flag 1 for generic names etc.).

Outputs include, one is a crosswalk that includes original patient id used to create the patient id hash and other is the hashed file without the original patient id. The crosswalk should be retained in a secured location to identify patients and link back to the original patient record. The hash file is a shareable file that can be shared with the aggregator (site aggregating data for disambiguation) based on the agreed protocol (e.g. sFTP etc). There may be additional output files based on the program language and selected output parameters. Please refer to Appendix B for each SQL and Java specific output files.

Optional Encryption: Sites may decide to further encrypt their hashed data file. Public key encryption is inefficient for large data sets, so a hybrid cryptosystem[[6]](#footnote-6) is used. The hash output file is encrypted using symmetric AES-256 encryption[[7]](#footnote-7) and the symmetric key is encrypted with RSA[[8]](#footnote-8) 2048-bit public key encryption. The aggregator should provide the RSA public key to participating sites for encrypting hash files (and retain private key pair securely) and the sites should send their shareable hash file as well as the encrypted symmetric key to the aggregator. Encryption is supported only in Java version.

Build 2 will allow the users to add data validation rules (e.g. patient data of birth should be after year 1900), generate exclusion flag (indicating do not match) for patients with generic names from the config file itself, and allow connection to the RDBMS to read patient data.

Note: For improved security, our next version may process the exclusion flag during hashing by not generating any hash values (except PIDHash, since Patient ID is required) in the output.

**Module 3: Disambiguation**

The disambiguation module matches patient records across and within sites and assigns matching records same global ID (also called Master ID). Matching is done on hashed composite identifiers using deterministic algorithm[[9]](#footnote-9). The aggregating site can be an Honest Broker or any participating site that does not have the Shared project salt.

**Matching Methodology**

Deterministic algorithms determine whether record pairs agree or disagree on a set of identifiers, where agreement on a given identifier is assessed as a discrete—“all-or-nothing”—outcome[[10]](#footnote-10). The match occurs on a set of identifiers that have been standardized, combined and hashed similarly across all participating sites (handled in Module 2: Hashing). When using composite identifiers (e.g., First Name + Last Name + Date of Birth + Last 4 SSN), equal weight is assigned to each data element and only when the entire composite ID matches, it is flagged as a match. In our current build, 10 composite identifiers are processed that allow syllogistic matches including, iterative match, hierarchical match (from more specific to more sensitive), and fuzzy match (partial name match). Also, see Appendix C, Table 1 for full list of rules. Deterministic algorithm is simple to understand, easy to implement and quality of matches can be improved greatly by improving data normalization techniques and increasing identifiers. Below are key match rules:

Rule 0: any hash <--> any hash

Rule 1: any hash with Full Name DOB SSN <--> any hash with Full Name DOB SSN

Rule 2: any hash with Full Name DOB <--> any hash with Full Name DOB

Rule 3: Hash 1 Full Name DOB SSN <--> Hash 1 Full Name DOB SSN

Rule 4: Hash 1 Full Name DOB SSN <--> Hash 2 Transposed Full Name DOB SSN

Rule 5: Hash 1 Full Name DOB SSN <--> Hash 5 Full Name Transposed DOB SSN

Rule 6: Hash 1 Full Name DOB SSN <--> Hash 9 Full Name DOB 1 Day SSN

Rule 7: Hash 1 Full Name DOB SSN <--> Hash 10 Full Name DOB 1 Year SSN

Rule 8: Hash 3 Full Name DOB <--> Hash 3 Full Name DOB

Rule 9: Hash 3 Full Name DOB <--> Hash 4 Transposed Full Name DOB

Rule 10: Hash 3 Full Name DOB <--> Hash 6 Full Name DOB 1 Year SSN

Rule 11: Hash 7 Partial Name DOB SSN <--> Hash 7 Partial Name DOB SSN

Rule 12: Hash 8 Partial Name DOB <--> Hash 8 Partial Name DOB

Notes:

* Full Name = First Name + Last Name
* Transposed Full Name = Last Name + First Name
* DOB = Date of Birth (MM/DD/YYYY)
* Transposed DOB = Transposed Date and Month in Date of Birth (DD/MM/YYYY)
* SSN = Last 4 Social Security Numbers
* Partial Name = Initial 3 characters in First Name + Last Name
* Rules 0, 1, and 2 are legacy rules. These are available for backward compatibility and will be phased out

The match rules can be combined to adjust the sensitivity and specificity of matching e.g., combining rules 3, 4, 5, 6, 7 increases the sensitivity relative to rule 3 and yet retains specificity of requiring SSN for the matches. See Appendix C, Table 1 on recommendations for combining rules.

**Theoretical Construct**

When two patients form a link by implementing any of the above rules, they form an edge. In a simple graph, with N patients, there are at most N(N-1)/2 edges[[11]](#footnote-11). The patients with a connecting edge are assigned same Global ID. Output from each rule is called a match set. The match sets can be cross combined for sequential iterative matching. Hierarchy matching can be implemented by selecting hashes that range from more specific to more sensitive e.g., Partial name match (fuzzy matching) with date of birth and without SSN is ranked more sensitive and with SSN is more specific. In hierarchal order, match rules 3; 4; 5; 8 &11 are more specific match sub-sets of match rules 8,11&12; 9; 10; 12 respectively.

**Operational Steps**

In Java, the match engine has 2 components:

1. A data load routine that loads the data files to a SQLite[[12]](#footnote-12) database table with pre-determined indexes. Using SQLite open source relational database system optimizes sorting of data
2. An assignment routine that evaluates the set of rules, searches for agreement, and assigns Global ID to patient records based on the rules premise

The engine allows flexible matching and patient records can be matched across match sets. In this case, each match set becomes a vector containing all patient nodes and previously non-matched patients are sequentially matched across vectors in the specified sequence e.g., Match on Rule 3 (more specific match requiring full name, date of birth and last 4 SSN) and then match previously non-matched patients on Rule 4 (more sensitive match allowing transposed name, date of birth and last 4 SSN). For a comprehensive list of recommended match set combinations, see Appendix C. The engine also evaluates the exclusion flag that was created during hashing and if the patient record has exclusion flag set to 1, the record will not be matched and will be assigned a unique global ID.

Appendix A: Salt engine

The salt engine generates unique encrypted salt file for each site in the project. To begin, each participating site needs to generate 2048-bit private-public key pair[[13]](#footnote-13). The private key should be stored in a secured location and never shared. The private key is an input in the hashing module and is used to decrypt the encrypted salt file. The public key should be sent to the salt generating site also termed as Key Master. The public key is used to encrypt the participating site’s salt file.

For new projects, the Key Master, will need below inputs:

1. The Project Name
2. Unique Site ID and name for each participating site
3. RSA 2048-bit public key file from each participating site (this should be a separate text file from each participating site)

There is also an option to add more sites to an ongoing project. For adding sites to an ongoing project, one of the existing participating site should take on the role (acting as Key Master) of generating the encrypted salt file and distributing it to the new site since they already have the encrypted salt file that can be opened with their private key. The acting Key Master will need 2 additional inputs:

1. Encrypted Salt file (file containing salt for project new site is being added to)
2. Private key that was part of the public-private key pair used to encrypt the existing participating site’s shared project Salt file

Additional inputs will be required when using web interface.

The output is an asymmetric encrypted salt file for each site id in the input file. The output file naming convention is ProjectName\_SiteID\_Date.txt

Each encrypted output file is a comma separated value (see Appendix D for Data Dictionary):

1. Site ID – unique ID for each site
2. Site Name – unique Name for each site
3. Private Salt – unique 13 (or more) character string for each site during a single salt generation instance
4. Shared Salt – same 13 (or more) character string for all sites on a project
5. Project ID – same for all sites on a project

The salt engine checks to ensure that the shared salt is different from private salts, and that all private salts are unique during that instance of salt generation. When adding, new sites to an ongoing project, since the acting Key Master will not have other sites’ private salts, there is a risk that newly added site could have private salt collision (i.e., same private salt as other site).

A.1 Instructions for using Java program

This is a standalone command line program for generated encrypted shareable salt file and does not require any web service connection.

System Requirements:

* Linux, macOS or Windows OS
* Java Runtime Environment (JRE) 1.8 or higher

Download or clone the program from Github (<https://github.com/linkja>). Pre-built JAR is available from the releases page (https://github.com/linkja/salt-engine/releases).

**Input Data**

1. Site definitions – comma separated value (csv) file with site ID, site name, file path to respective site’s public key

**Executing the program**: From command line, run the executable JAR file using the standard Java command: java -jar SaltEngine-1.0-jar-with-dependencies.jar

The program is expecting a minimum of two parameters:

--projectName = Project Name

--siteFile = Path to a file containing the site definitions

There are additional optional parameters that may be specified to generate random private salt for additional sites after the initial random project salt:

--encSaltFile = path to encrypted Salt file containing the shared project salt to be used

--privateKey = Path to private key (part of the public-private key pair used to encrypt salt file)

Example:

Navigate to the directory containing jar file and run below command:

java -jar SaltEngine-1.0-jar-with-dependencies.jar -- projectName Linkja\_project1 --siteFile ./siteKey/project1\_sitedefinitions

Basic Troubleshoot:

* Data must not be longer than 245 bytes – RSA encryption cannot encrypt large data. To proceed, reduce the length of Site ID, Site Name and/or Project Name (also, minimum length of project and private salt is 13 characters each)
* Cannot find jar file – From command line, navigate to the folder where jar file is located and then run the Java command or specify full path to the JAR file

Appendix B: De-identification

The hashing module generates composite patient identifier hashes. This module is available as a Java program and SQL-batch Scripts. Although each program has different implementation, the hash outputs will be same for same data source from both programs. The sites can choose from either based on their preferences. Patient identifiers data requirement for both the modules are same. Following inputs are required for hashing:

1. Patient data (see Appendix D: Data Dictionary #2 Patient Data)
2. Encrypted salt file (see Appendix A to generate file and Appendix D for data dictionary)
3. Private key file (file containing private key, part of the public-private key pair used to the generate encrypted salt file)
4. Private date any random date (as MM/DD/YYYY)

Optional input file:

1. Aggregator’s public key file (Public key provided by aggregator to encrypt the symmetric key used to encrypt the hash output file)

The patient source data should meet these requirements:

1. Each site should have a unique patient ID for each record (e.g. use system level identifier instead of facility level[[14]](#footnote-14))
2. Patient ID, patient first name, patient last name, and patient date of birth are required fields
3. Social Security Number and exclusion flag are optional fields
4. Appendix D includes data dictionary for the input data

Outputs: There is 1 main output and several optional outputs depending on program version (Java or SQL-batch Scripts) and selected parameters:

* Shareable Hash file (csv file for SQL and Java. If optional parameter encryption is on in Java, encrypted hash file is generated)
* Shareable encrypted key (in Java, if optional parameter encryption is on, then a file containing symmetric key encrypted with aggregator’s public key is generated)
  + Java uses AES-256-GCM encryption to encrypt the hash file
* Crosswalk (SQL table, in Java csv file only if optional parameter is true)
* Invalid data (SQL query, in Java csv file with error description)

B.1 Instructions for using Java program

This is a standalone command line program for de-identifying and hashing patient data and does not require any web service connection.

System Requirements:

* Linux, macOS or Windows OS
* Java Runtime Environment (JRE) 1.8 or higher

Download or clone the program from Github (<https://github.com/linkja>). Pre-built JAR is available from the releases page (<https://github.com/linkja/linkja-hashing/releases>).

**Input Data**

1. Patient data – can be comma separated value (csv) file or delimited text file with headers (e.g., |) (see Appendix D #2 Patient Data for data dictionary)
2. Encrypted salt file (see Appendix A to generate file and Appendix D for data dictionary)
3. Private Key file (file containing private key, part of the public-private key pair used to the generate encrypted salt file)
4. Private date (as MM/DD/YYYY)
5. Aggregator’s public key file (optional, if hash file is to be encrypted, the aggregator will provide this file)

**Executing the program**: From command line, run the executable JAR file using the standard Java command: java -jar Hashing-1.0-jar-with-dependencies.jar

The program has two modes that can be invoked - the primary one (enabled with `--hashing`) will perform the hashing operations on an input file. The second mode (which can be used by itself or with hashing) is enabled with `--displaySalt`. This will decrypt and display the contents of the encrypted salt file.

1. Hashing Mode Usage: `java -jar Hashing-1.0-jar-with-dependencies.jar --hashing`

The program is expecting a minimum of four parameters:

--privateDate = The private date (as MM/DD/YYYY)  
 --privateKey = Path to the private key file  
 --patientFile = Path to the file containing patient data  
 --saltFile = Path to encrypted salt file

There are additional optional parameters that may be specified:

--outDirectory = The base directory to create output. If not specified, will use the current directory.

--delimiter =The delimiter used within the patient data file. Uses a comma "," by default.

--writeUnhashed (true/false) = Writes out the original unhashed data in the result file (for debugging). false by default.

--encryptionKey = Path to a public key file to then be used to encrypt hashed output

Example 1 (required parameters specified):

Navigate to the directory containing jar file and run below command:

java -jar Hashing-1.0-jar-with-dependencies.jar --hashing --privateKey ./keys/private.key --saltFile ./data/project1\_stalt\_encrypted.txt --patientFile ./data/project1\_patients.csv --privateDate 01/01/2018

Example 2 (pipe delimited file as input, and specify the output directory. Also includes writing out the unhashed data for debugging):

java -jar Hashing-1.0-jar-with-dependencies.jar --hashing --privateKey ./keys/private.key --saltFile ./data/project1\_stalt\_encrypted.txt --patientFile ./data/project1\_patients.csv --privateDate 01/01/2018 --outDirectory ./data/output/ --delimiter | --writeUnhashed

When program executes successfully in hashing mode, there will be message with data rows processed and execution time. Depending on the parameters provided during execution, there will be 4-5 outputs:

Output 1: Shareable hash file – this file contains de-identified hash data and should be sent to the aggregator for matching. The file name is hashes\_siteid\_projectid\_datetime.csv (Appendix D). If the encrypted option, file name is enc\_hashes\_siteid\_projectid\_datetime.csv

Output 2: Crosswalk between local ID and hashed patient ID - this file contains crosswalk and should be kept in a secured location and shouldn’t be shared

Output 3: Invalid data file – this file contains records that did not meet valid data criteria described previously and the error description. Once the errors are fixed, execute the program again to hash the records. This file should not be shared.

Output 4: Review file – if the optional parameter -unhashed was true during execution, then a review file with normalized data and hashes is generated. This file is for review only and should not be shared.

Output 5: Shareable encrypted Key file – if optional encryption Key was provided, this file is generated and should be shared with the aggregator. File is encrypted and contains the Symmetric key used encrypt the hash file.

Display Salt Mode

Usage: `java -jar Hashing-1.0-jar-with-dependencies.jar --displaySalt`

The program is expecting a minimum of two parameters:

--privateKey = Path to the private key file

--saltFile = Path to encrypted salt file

Example:

java -jar Hashing-1.0-jar-with-dependencies.jar –displaySalt --privateKey ./keys/private.key --saltFile ./data/project1\_stalt\_encrypted.txt

Notes:

* Where files are used for input, they can be specified as a relative or absolute path. If jar file and all the inputs are not in same directory, provide full directory and file paths

**Only for users interested in compiling their own JAR files:**

**Compiling the program:** linkja-hashing was built using Java JDK 1.8 (specifically OpenJDK). It can be opened from within an IDE like Eclipse or IntelliJ IDEA and compiled, or compiled from the command line using Maven. Compiling jar file requires internet connection to download the dependencies required to create the jar file.

*mvn clean package*

This will compile the code, run all unit tests, and create an executable JAR file under the .\target folder with all dependency JARs included. The JAR will be named something like Hashing-1.0-jar-with-dependencies.jar.

**Editing the configurations:** Users can configure several parameters

1. Navigate to . \linkja-master\Hashing\src\main\resources

config.properties

recordExceptionMode = Generate

runNormalizationStep = true (normalizes the data as per rules listed in the document)

workerThreads = 7 (adjust the threads based on available RAM)

batchSize = 40000 (adjust the batch size based on available RAM)

minSaltLength = 13 (if the salt length does not meet this criteria, the app will throw an error. We recommend minimum 13 characters)

2. Navigate to .\linkja-master\Hashing\src\main\resources\configuration

a. edit canonical-header-names: add the alias header names to match the header values in the data

b. edit suffixes: add or remove suffixes based on your data. There should be 1 suffix per row. It is space sensitive

c. edit prefixes: add or remove suffixes based on your data. There should be 1 suffix per row. It is space sensitive

After editing the configurations, re-compile the jar file for new configurations to take effect.

**Basic Troubleshoot**:

* Setting Java\_home –http://sbndev.astro.umd.edu/wiki/Finding\_and\_Setting\_JAVA\_HOME
* Cannot find jar file – From command line, navigate to the folder where jar file is located and then run the Java command or specify full path to the JAR file
* Windows users: if the directory or file name contains spaces, use qualified path names with quotes e.g. "C:\directory Path\file.txt"
* Linux users: if the file was created on windows, convert the file e.g. dos2unix delimited\_text\_file.txt

B.2 Instructions for using SQL program

This program works with MS SQL Server 2012/2014 and Windows 2010+. Download or clone the program from Github (<https://github.com/linkja/SQL-hashing>)

1. Edit the config file in the master folder with your institution’s database & field names. Below are sample values (read below instructions on configuration for more details):

Server=data.local (or 12.69.100.120)

Database=CAPriCORN\_CDM\_V4

schema=htotest

sourceTable=CAPriCORN\_CDM\_V4.dbo.CAP\_PATIENTDATA

patientid=EnterpriseID

name1=FIRST

name2=LAST

dob=DOB

ssn=SSN

exclusion=1

temptablename=#temptabletostorecleanDATA

hashTable=dbo.hthisistotest

privateDate=01/31/1960

encryptedSaltfile="C:\Users\kdoshi\Desktop\CRU-PPRL-hashing-application-master\CRU-PPRL-hashing-application-master\sitename\_project\_1\_salt.txt"

privateKeyfile="C:\Users\kdoshi\Desktop\CRU-PPRL-hashing-application-master\CRU-PPRL-hashing-application-master\privateKey\_project.txt"

1. Run hash\_setup.bat file by double clicking on it. This sets up stored procedures and functions in SQL that will read, normalize, concatenate and hash the data (this is one-time activity). This may take a minute to run depending on the resources. When the program compeletes, the message of cmd is displayed “Completed successfully. Press the ENTER key to exit...:”
2. Run hash\_run\_site.bat by double clicking on it. This generates a SQL table and a csv file with the hashed values (run it as many times as you want to create the hashes). This may take some time to run depending on your resources and size of data. When the program completes, the message on cmd is displayed “Completed successfully. Press the ENTER key to exit...: ”
3. Review the outputs: SQL crosswalk table, shareable csv hash file, and Log folders.
4. To view invalid data, run the ‘Review invalid data.sql’ from Extras folder
5. Additional:
   1. The HashSetup SQL scripts can be edited to include additional suffixes, prefixes
   2. InsertIntoHash SQL scripts can be edited to define the conditions for exclusion flag in the query itself (instead of pre-processing the data)

7. Extras directory contains additional batch scripts to review the encrypted salt file contents. It is advisable to review the contents of Salt File and ensure that the site received correct Salt file by verifying site name and site id. Run the todecrypt\_SaltFile\_foruserreview batch file from Extras folder and follow the prompts.

**Instructions on Configuration:**

1. Enter server name or server and instance name as servername or servername/instance
2. To store set up functions, enter database name and schema name
3. Enter demographic table name as sourceTable=database.schema.table or a sql sub-query
4. Enter field names from the table e.g. patientid=PersonID, name1=firstname, name2=lastname, dob=birthdate, ssn=social\_security\_number, exclusion=flag1or0
   * If SSN is not to be included in hashing, then keep ssn=0
   * If exclusion flag is not to be included in hashing, then keep exclusion=0
5. Provide a temporary table name to store clean data e.g. temptablename=#temptabletostorecleanDATA (# followed by any name)
6. Provide a table name to store hashes e.g. hashTable=database.schema.table
   * If there is an existing table with same name, that table is replaced
7. each site should select a random date (MM/DD/YYYY) for privateDate. This should not be shared with anyone
8. provide the file name and paths to encryptedSaltfile and your privateKeyfile e.g. "C:\Users\sitename\_project\_1\_salt.txt"
9. Please ensure that patientid is unique for each record and there are no duplicates

**SQL Scripts:**

SQL queries can be found in the SQL Files directory:

1. Hash Setup – Includes all the functions used for data normalization and hashing

2. Insert into Hash – Includes all the hashing process

Note: SQL queries can be edited and run as stand-alone scripts if all the parameter values are available as distinct values (i.e., non-encrypted site details, project name, and salt values are required)

Appendix C: Disambiguation

The aggregating site uses disambiguation module to match patient records across sites and to assign matching records same global ID using specified deterministic algorithms[[15]](#footnote-15). This module is available as a Java program and works with SQLite. It includes complete implementation of several matching rules and is available as a command line program as well as user-friendly interface (GUI).

**Matching Rules**

There are several matching rules available that range from most specific to most sensitive. Based on the project requirements, each rule can be implemented either by itself or as combination of rules.

Table1: Complete syllogistic matching set rules available in Java:

|  |  |  |
| --- | --- | --- |
| **Rule No. (Match set)** | **Composite identifiers (Hashed)** | **Match Rule Description** |
| 0\* | All hashes with each other | All hashes with each other |
| 1\* | hash1,hash2,hash5,hash9,hash10 with hash1,hash2,hash5,hash9,hash10 | All Full name, dob & SSN |
| 2\* | hash3,hash4,hash6 with hash3,hash6,hash6 | All Full name & dob |
| 3 | hash1 with hash1 | Full name, dob & SSN |
| 4 | hash1 with hash2 | Full transposed name, dob & SSN |
| 5 | hash1 with hash5 | Full name, transposed dob & SSN |
| 6 | hash1 with hash9 | Full name, day diff dob & SSN |
| 7 | hash1 with hash10 | Full name, year diff dob & SSN |
| 8 | hash3 with hash3 | Full name, dob |
| 9 | hash3 with hash4 | Full transposed name, dob |
| 10 | hash3 with hash6 | Full name, transposed dob |
| 11\*\* | hash7 with hash7 | Partial name, dob & SSN\* |
| 12\*\* | hash8 with hash8 | Partial name, dob\* |

\*Rules 0,1,2 are legacy match rules carried forward for backward compatibility

\*\*Rules 11 and 12 are fuzzy matches allowing first 3 initials of first name combined with rest of the elements as specified in description

The rules can be combined to form more complex algorithms. Recommended cross match set rule combinations:

3 – most specific

3,4,5,6,7

3,4,5,6,7,11

8,9,10

12

12,9,10,6,7 – most sensitive

C.1 Instructions for using Java program

This is a standalone command line program, works with SQLite (open source database management system) and does not require any web service connection.

System Requirements:

* Linux, macOS or Windows OS
* SQLite browser available from https://nightlies.sqlitebrowser.org/latest [[16]](#footnote-16)
* Java Runtime Environment (JRE) 1.8 or higher
* Patient matching and global ID assignment is a resource intensive activity. Depending on the data size optimum RAM size may be required

Download or clone the program from Github (<https://github.com/linkja>). Pre-built JAR is available from the releases page (<https://github.com/linkja/linkja-matching/releases>).

**Input File**

Input file should be csv or | delimited text file with Site ID, Project ID and PIDHash as required fields. Please see Appendix D #3 Shareable Hash Output for input file data dictionary and Appendix B for generating the shareable hash output file.

**Program Set-up**

**1. Create project directory and sub-directories (do this for each project)**

Aggregator could have multiple projects with data linkage, to ensure integrity of matching, a separate root directory is recommended for each project with below sub-directory set up:

1. Create a project specific root directory:

./Project Name

1. Create config sub-directory within project directory

./ProjectName/Config

1. Create data and other sub-directories within project directory

./ProjectName/data

./ProjectName/data/Input – all hash input files should be here

./ProjectName/data/Processed – contains all log and processed hash files

./ProjectName/data/Output – all site reports will be saved here

Note: configuration files (global-match.properties.properties and global-match-globalId.txt) should be downloaded from above github and saved in the Config directory

**2. Create SQLite database and tables (use SQLite browser to do this)**

1. In directory ./Project Name/data, create GlobalMatchData database
2. In GlobalMatchData database, create 3 tables and write changes to the database:
   1. InclusionPatients – table contains all patient records that are to be matched (i.e., Exclusion field is not 1)
   2. ExclusionPatients – table contains all patient records that are not to be matched (i.e., exclusion field is 1). These records are assigned unique global ID
   3. GlobalMatch – table contains all records from InclusionPatients table with Global ID assigned from user specified algorithm

CREATE TABLE IF NOT EXISTS InclusionPatients (

id integer PRIMARY KEY,

globalId integer,

siteId text,

projectId text,

pidhash text NOT NULL,

hash1 text,

hash2 text,

hash3 text,

hash4 text,

hash5 text,

hash6 text,

hash7 text,

hash8 text,

hash9 text,

hash10 text,

exclusion text

);

CREATE TABLE IF NOT EXISTS ExclusionPatients (

id integer PRIMARY KEY,

globalId integer,

siteId text,

projectId text,

pidhash text NOT NULL,

hash1 text,

hash2 text,

hash3 text,

hash4 text,

hash5 text,

hash6 text,

hash7 text,

hash8 text,

hash9 text,

hash10 text,

exclusion text

);

CREATE TABLE IF NOT EXISTS GlobalMatch (

id integer PRIMARY KEY,

globalId integer,

siteId text,

projectId text,

pidhash text NOT NULL,

hash1 text,

hash2 text,

hash3 text,

hash4 text,

hash5 text,

hash6 text,

hash7 text,

hash8 text,

hash9 text,

hash10 text,

exclusion text

);

1. Create index on the GlobalMatch table as below and write the changes to the database:

CREATE INDEX match0 ON GlobalMatch (hash1,hash2,hash3,hash4,hash5,hash6,hash7,hash8,hash9,hash10,id);

CREATE INDEX match1 ON GlobalMatch (hash1,hash2,hash5,hash9,hash10,id);

CREATE INDEX match2 ON GlobalMatch (hash3,hash4,hash6,id);

CREATE INDEX match3 ON GlobalMatch (hash1,id);

CREATE INDEX match4 ON GlobalMatch (hash1,hash2,id);

CREATE INDEX match5 ON GlobalMatch (hash1,hash5,id);

CREATE INDEX match6 ON GlobalMatch (hash1,hash9,id);

CREATE INDEX match7 ON GlobalMatch (hash1,hash10,id);

CREATE INDEX match8 ON GlobalMatch (hash3,id);

CREATE INDEX match9 ON GlobalMatch (hash3,hash4,id);

CREATE INDEX match10 ON GlobalMatch (hash3,hash6,id);

CREATE INDEX match11 ON GlobalMatch (hash7,id);

CREATE INDEX match12 ON GlobalMatch (hash8,id);

CREATE INDEX pidindex ON GlobalMatch (pidhash,siteId,projectId,id);

COMMIT;

1. Create a view on GlobalMatch table and write changes to the database:

CREATE VIEW report1 AS

SELECT siteId, projectId, pidhash, globalId

FROM GlobalMatch;

**Executing the program**: From command line, run the executable JAR file using the standard Java command: java -jar Matching-0.1-jar-with-dependencies.jar --directory (project root directory). The program has three processing steps that can be invoked:

1. Load Data: this command loads the hash data files from the input folder to SQLite database. Records with exclusion flag 1 are loaded to ExclusionPatients table and all the other records are loaded to InclusionPatients and GlobalMatch table. The processed files are moved to Processed directory and a log file is created in the Processed directory. Usage: `java -jar Matching-0.1-jar-with-dependencies.jar --directory --load`

The program is expecting a minimum of three parameters:

--directory = project root directory

--prefix = beginning few characters of the filename (it is advisable to have all the filenames with same prefix so as to reduce the number of times this step needs to be executed)  
 --suffix = file extension (to process all file types at same time, use "")

Example:

java -jar Matching-0.1-jar-with-dependencies.jar --directory "O:\linkja-matching-master\project 1" --load --prefix hashes --suffix .csv

If there are files with different prefixes, change the prefix and re-execute the command until all the files in the input folder are processed. SiteID, ProjectID, PIDHash and Exclusion are required fields. If these are missing, the program will store the record in a file with .bad extension in Input directory and continue processing other records. As per the agreed protocol, the records in the file can either be shared back with the site that sent the data for data check or can be ignored.

2. Match Data: this command matches the hash data in GlobalMatch table. Usage: `java -jar Matching-0.1-jar-with-dependencies.jar --directory --match`

The program is expecting a minimum of three parameters:

--directory = project root directory

--match = rule number to use for matching. To run multiple rules, provide comma separated values e.g., 3, 4, 5. For each comma separated rule, the application matches previously un-matched patient records iteratively. However, all the rules to be applied should be provided prior to the run.   
 --seed = starting seed for Global ID

Example:

java -jar Matching-0.1-jar-with-dependencies.jar --directory "O:\linkja-matching-master\project 1" --match 9 --seed 1

3. Report Data: this command combines the data from exclusion table and GlobalMatch table and generates comma delimited shareable Site specific text files with Global ID and Patient ID Hash crosswalk. Usage: `java -jar Matching-0.1-jar-with-dependencies.jar --directory --report`

The program is expecting a minimum of two parameters:

--directory = project root directory

--report = Site ID (for all sites, use all)

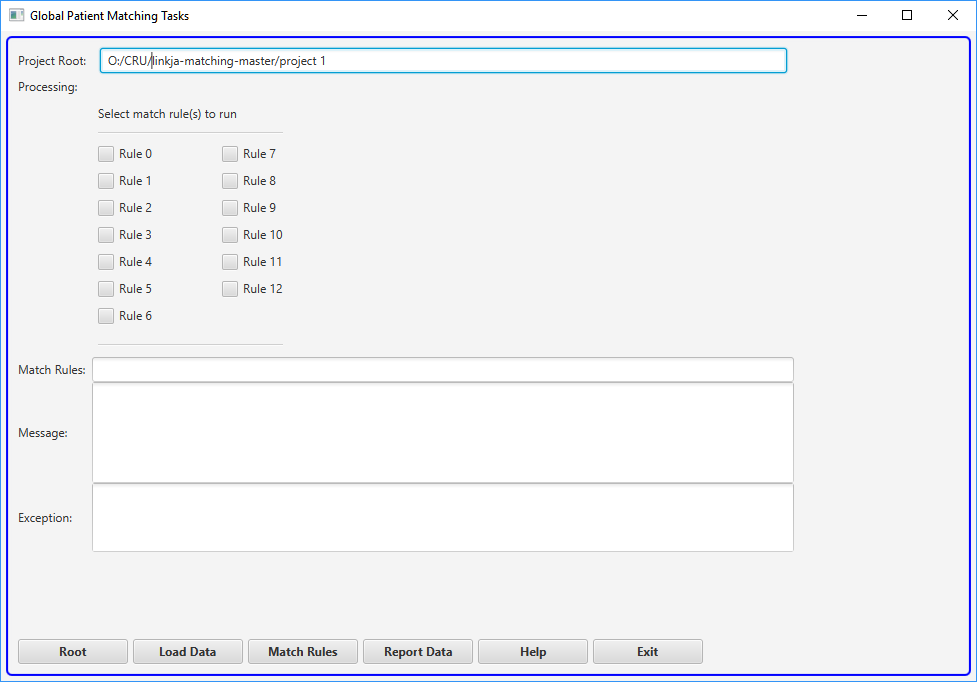
Example:

java -jar Matching-0.1-jar-with-dependencies.jar --directory "O:\linkja-matching-master\project 1" –report Site101

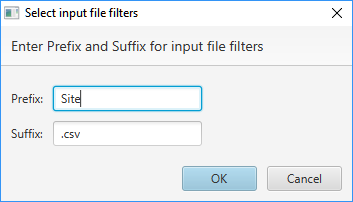
**GUI Mode**: For smaller data sets, the program can be executed in user-friendly GUI mode. Usage: java -jar Matching-0.1-jar-with-dependencies.jar --directory (project root directory) --GUI

Underlying load, match and report data processes are similar to the full command line version.

Example:

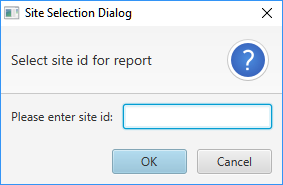


1. To load the data, click load data and enter prefix and suffix, and click OK



2. To match data, select the rules and click Match Rules

3. To generate shareable site specific report, enter side id and click OK (for all sites, enter all)



**Only for users interested in compiling their own JAR files:**

**Compiling the program**

linkja-disambiguation was built using Java JDK 1.8 (specifically OpenJDK). It can be opened from within an IDE like Eclipse or IntelliJ IDEA and compiled, or compiled from the command line using Maven.

*mvn clean package*

This will compile the code, run all unit tests, and create an executable JAR file under the .\target folder with all dependency JARs included. The JAR will be named something like Matching-0.1-jar-with-dependencies.

**Additional Configuration:** Config directory includes two files that are required by the program. Changing these files is not recommended.

1. global-match.properties:

a. Change default directories and database name as appropriate

b. Specify file prefix and suffix for load and the matching rule to apply (ignores these values when running from command line or GUI). To run multiple rules, provide comma separated values e.g., to run match full name, date of birth, ssn with itself, transposed names and transposed date of birth hashes, MatchingRules=3,4,5. For each comma separated rule, the application matches previously un-matched patient records iteratively. However, all the rules to be applied should be provided prior to the run.

2. global-match-globalId: Provide the seed (starting number) from where the Global IDs will be assigned (ignores these values when running from command line or GUI).

**Basic Troubleshoot**:

* Set Java\_home – http://sbndev.astro.umd.edu/wiki/Finding\_and\_Setting\_JAVA\_HOME
* Cannot find JAR file – From command line, navigate to the folder where jar file is located and then run the Java command or specify full path to the JAR file
* SQLite\_Busy message – write changes to GlobalMatchData database, close SQLite browser, and ensure no other processes are running on the database
* Root directory not found message – ensure that the root directory specified is points to the correct project and all the data and config sub-directories are present
* No input files found in directory message – ensure the files exist in the directory and suffix and prefix arguments match the files in the input directory.

Appendix D: Data Dictionary

1. RSA 2048-bit encrypted salt file (comma separated text file)

|  |  |  |
| --- | --- | --- |
| Column | Data Type | Description |
| Site ID | string or number | Unique Site ID\* |
| Site Name | string | Unique Site Name |
| Private Salt\*\* | string | Unique salt for each site |
| Shared Salt\*\* | string | Salt shared across all sites in the project / study |
| Project Name | string | Project Name |

\*If there are different data sources within a site (e.g. lab system and registration), consider using different site IDs or create a unique record for all patients within site

\*\*Length of Salt is 13 or more characters

2. Patient data

For Java: csv or text delimited file (headers required)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Patient ID | string or number | Unique patient identifier\* |
| 2 | First Name | string | Patient first name |
| 3 | Last Name | string | Patient last name |
| 4 | DOB | string | Date of birth\*\*\*\* |
| 5\*\* | SSN | string | Social Security Number\*\*\* |
| 6\*\* | Exclusion | string or number | 0 or blank – Allow patients to be matched  1 – Excludes patients from disambiguation and assigns them unique global ID |

\*Each patient record should have a unique ID

\*\* Fields 5 and 6 are optional fields, if the SSN is missing, hashes that require SSN (e.g. hash1 - fnamelnamedobSSN) will remain blank. If exclusion flag is missing, then it is assumed to be 0

\*\*\* The application can handle last 4 SSN as well as full Social Security Number (only last 4 numbers get used in hashing)

\*\*\*\*The application can handle several date formats YYYY-MM-DD (e.g. 1960-12-31), YYYYMMDD (e.g. 19601231), MM/DD/YYYY (e.g. 31/12/1960)

For SQL: database table

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Patient ID | varchar or int | Unique patient identifier\* |
| 2 | First Name | varchar | Patient first name |
| 3 | Last Name | varchar | Patient last name |
| 4 | DOB | datetime | Date of birth in DBMS |
| 5\*\* | SSN | varchar | Social Security Number\*\*\* |
| 6\*\* | Exclusion | int | 0 or blank – Allow patients to be matched  1 – Excludes patients from disambiguation and assigns them unique global ID |

\*Each patient record should have a unique ID

\*\*Fields 5 and 6 are optional fields, if the SSN is missing, hashes that require SSN (e.g. hash1 - fnamelnamedobSSN) will remain blank. If exclusion flag is missing, then it is assumed to be 0

\*\*\* The application can handle last 4 SSN as well as full Social Security Number (only last 4 numbers get used in hashing)

3. Shareable Hash Output. This file should be shared with the aggregator for matching and global ID assignment

For Java and SQL: csv (with headers)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Site ID | String | Site ID |
| 2 | Project ID | String | Project ID |
| 3 | PIDHASH | String | Patient ID + Site ID + Date Offset (Private Date and DOB) |
| 4 | hash1 | String | First Name + Last Name + DOB + L4 SSN |
| 5 | hash2 | String | Last Name + First Name + DOB + L4 SSN |
| 6 | hash3 | String | First Name + Last Name + DOB |
| 7 | hash4 | String | Last Name + First Name + DOB |
| 8 | hash5 | String | First Name + Last Name + Transposed DOB + L4 SSN |
| 9 | hash6 | String | First Name + Last Name + Transposed DOB |
| 10 | hash7 | String | 3 Initials First Name + Last Name + L4 SSN |
| 11 | hash8 | String | 3 Initials First Name + Last Name + DOB |
| 12 | hash9 | String | First Name + Last Name + DOB +1D + L4 SSN |
| 13 | hash10 | String | First Name + Last Name + DOB +1Y + L4 SSN |
| 14 | Exclusion | String | 0 – Allow patients to be matched  1 – Excludes patients from disambiguation and assigns them unique global ID |

DOB=date of birth (YYYY-MM-DD)

Transposed DOB = Month and Date Transposed in date of birth (YYY-DD-MM)

1D = 1 day offset in date of birth

1Y = 1 year offset in date of birth

L4 SSN = Last 4 Social Security Numbers

Fields 3 – 13 are SHA512[[17]](#footnote-17) hashes (128 hexadecimal characters)

4. Hash Crosswalk: Hash Output and Local Site-Patient ID. This data should not be shared since it contains original Patient ID

For Java: csv (with headers)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Patient ID | string | Site’s original Patient ID |
| 2 | PIDHASH | string | Patient ID + Site ID + Date Offset (Private Date and Date of Birth) |

For SQL: database table (with headers)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Site ID | varchar | Site ID |
| 2 | Project ID | varchar | Project ID |
| 3 | Internal ID | varchar or int | Original Patient ID |
| 3 | PIDHASH | varchar(128) | Patient ID + Site ID + Date Offset (Private Date and DOB) |
| 4 | hash1 | varchar(128) | First Name + Last Name + DOB + L4 SSN |
| 5 | hash2 | varchar(128) | Last Name + First Name + DOB + L4 SSN |
| 6 | hash3 | varchar(128) | First Name + Last Name + DOB |
| 7 | hash4 | varchar(128) | Last Name + First Name + DOB |
| 8 | hash5 | varchar(128) | First Name + Last Name + Transposed DOB + L4 SSN |
| 9 | hash6 | varchar(128) | First Name + Last Name + Transposed DOB |
| 10 | hash7 | varchar(128) | 3 Initials First Name + Last Name + L4 SSN |
| 11 | hash8 | varchar(128) | 3 Initials First Name + Last Name + DOB |
| 12 | hash9 | varchar(128) | First Name + Last Name + DOB +1D + L4 SSN |
| 13 | hash10 | varchar(128) | First Name + Last Name + DOB +1Y + L4 SSN |
| 14 | Exclusion | int | 0 – Allow patients to be matched  1 – Excludes patients from disambiguation and assigns them unique global ID |

5. Invalid data: Output file with invalid records that were not processed during hashing. This file is for review only and should not be shared since it contains PHI

For Java: csv (with headers)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Row Number | Number | Row number in the file containing invalid data |
| 2 | Patient ID | string or number | Unique patient identifier |
| 3 | First Name | string | Patient first name |
| 4 | Last Name | string | Patient last name |
| 5 | DOB | string | Date of birth |
| 6 | SSN | string | Last 4 Social Security Number |
| 7 | Error\_description | string | Detailed error description |

For SQL: database table (run the invalid data sql script from Extras folder)

6. Review data: In Java, if the optional parameter for unhashed=True, it generates file containing normalized and hashed data. This file is for review only and should not be shared since it contains PHI

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Site ID | String | Site ID |
| 2 | Project ID | String | Project ID |
| 3 | Patient ID | string or number | Original Unique patient identifier |
| 4 | First Name | string | Normalized Patient first name |
| 5 | Last Name | string | Normalized Patient last name |
| 6 | DOB | string | Normalized Date of birth |
| 7 | SSN | string | Normalized Last 4 Social Security Number |
| 8 | PIDHASH | String | Patient ID + Site ID + Date Offset (Private Date and DOB) |
| 9 | hash1 | String | First Name + Last Name + DOB + L4 SSN |
| 10 | hash2 | String | Last Name + First Name + DOB + L4 SSN |
| 11 | hash3 | String | First Name + Last Name + DOB |
| 12 | hash4 | String | Last Name + First Name + DOB |
| 13 | hash5 | String | First Name + Last Name + Transposed DOB + L4 SSN |
| 14 | hash6 | String | First Name + Last Name + Transposed DOB |
| 15 | hash7 | String | 3 Initials First Name + Last Name + L4 SSN |
| 16 | hash8 | String | 3 Initials First Name + Last Name + DOB |
| 17 | hash9 | String | First Name + Last Name + DOB +1D + L4 SSN |
| 18 | hash10 | String | First Name + Last Name + DOB +1Y + L4 SSN |
| 19 | Exclusion | String | 0 – Allow patients to be matched  1 – Excludes patients from disambiguation and assigns them unique global ID |

7. InclusionPatients: SQLite table

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Site ID | String | Site ID |
| 2 | Project ID | String | Project ID |
| 3 | Global ID | Integer | Always set to 0 |
| 4 | Patient ID | string | Original Unique patient identifier |
| 5 | First Name | string | Normalized Patient first name |
| 6 | Last Name | string | Normalized Patient last name |
| 7 | DOB | string | Normalized Date of birth |
| 8 | SSN | string | Normalized Last 4 Social Security Number |
| 9 | PIDHASH | String | Patient ID + Site ID + Date Offset (Private Date and DOB) |
| 10 | hash1 | String | First Name + Last Name + DOB + L4 SSN |
| 11 | hash2 | String | Last Name + First Name + DOB + L4 SSN |
| 12 | hash3 | String | First Name + Last Name + DOB |
| 13 | hash4 | String | Last Name + First Name + DOB |
| 14 | hash5 | String | First Name + Last Name + Transposed DOB + L4 SSN |
| 15 | hash6 | String | First Name + Last Name + Transposed DOB |
| 16 | hash7 | String | 3 Initials First Name + Last Name + L4 SSN |
| 17 | hash8 | String | 3 Initials First Name + Last Name + DOB |
| 18 | hash9 | String | First Name + Last Name + DOB +1D + L4 SSN |
| 19 | hash10 | String | First Name + Last Name + DOB +1Y + L4 SSN |
| 20 | Exclusion | String | 0 – Allow patients to be matched |

8. ExclusionPatients: SQLite table

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Site ID | String | Site ID |
| 2 | Project ID | String | Project ID |
| 3 | Global ID | Integer | Always set to 0 |
| 4 | Patient ID | string | Original Unique patient identifier |
| 5 | First Name | string | Normalized Patient first name |
| 6 | Last Name | string | Normalized Patient last name |
| 7 | DOB | string | Normalized Date of birth |
| 8 | SSN | string | Normalized Last 4 Social Security Number |
| 9 | PIDHASH | String | Patient ID + Site ID + Date Offset (Private Date and DOB) |
| 10 | hash1 | String | First Name + Last Name + DOB + L4 SSN |
| 11 | hash2 | String | Last Name + First Name + DOB + L4 SSN |
| 12 | hash3 | String | First Name + Last Name + DOB |
| 13 | hash4 | String | Last Name + First Name + DOB |
| 14 | hash5 | String | First Name + Last Name + Transposed DOB + L4 SSN |
| 15 | hash6 | String | First Name + Last Name + Transposed DOB |
| 16 | hash7 | String | 3 Initials First Name + Last Name + L4 SSN |
| 17 | hash8 | String | 3 Initials First Name + Last Name + DOB |
| 18 | hash9 | String | First Name + Last Name + DOB +1D + L4 SSN |
| 19 | hash10 | String | First Name + Last Name + DOB +1Y + L4 SSN |
| 20 | Exclusion | String | 1 – Excludes patients from disambiguation and assigns them unique global ID |

9. GlobalMatch: SQLite table

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Site ID | String | Site ID |
| 2 | Project ID | String | Project ID |
| 3 | Global ID | Integer | Global ID assigned after running match rules |
| 4 | Patient ID | string | Original Unique patient identifier |
| 5 | First Name | string | Normalized Patient first name |
| 6 | Last Name | string | Normalized Patient last name |
| 7 | DOB | string | Normalized Date of birth |
| 8 | SSN | string | Normalized Last 4 Social Security Number |
| 9 | PIDHASH | String | Patient ID + Site ID + Date Offset (Private Date and DOB) |
| 10 | hash1 | String | First Name + Last Name + DOB + L4 SSN |
| 11 | hash2 | String | Last Name + First Name + DOB + L4 SSN |
| 12 | hash3 | String | First Name + Last Name + DOB |
| 13 | hash4 | String | Last Name + First Name + DOB |
| 14 | hash5 | String | First Name + Last Name + Transposed DOB + L4 SSN |
| 15 | hash6 | String | First Name + Last Name + Transposed DOB |
| 16 | hash7 | String | 3 Initials First Name + Last Name + L4 SSN |
| 17 | hash8 | String | 3 Initials First Name + Last Name + DOB |
| 18 | hash9 | String | First Name + Last Name + DOB +1D + L4 SSN |
| 19 | hash10 | String | First Name + Last Name + DOB +1Y + L4 SSN |
| 20 | Exclusion | String | 0 – Allow patients to be matched |

10. Shareable Global ID Output: This file contains Global ID and Site-Patient ID and can be shared back with the respective sites. Precaution should be taken so that the site receives only their data file and no other sites’ data

comma separated text file (with headers)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Site ID | String | Site ID |
| 2 | Project ID | String | Project ID |
| 3 | Global ID | Integer | Global ID assigned after running match rules |
| 4 | Patient ID | string | Original Unique patient identifier |

**Pending features:**

* Salt engine – specifiying project salt for additional sites is pending
* Decrypt hash files (development in progress)
* Index page documentation

1. https://en.wikipedia.org/wiki/Public-key\_cryptography [↑](#footnote-ref-1)
2. https://en.wikipedia.org/wiki/Secure\_Hash\_Algorithms [↑](#footnote-ref-2)
3. https://en.wikipedia.org/wiki/Public-key\_cryptography [↑](#footnote-ref-3)
4. http://www.hl7.eu/refactored/segPID.html [↑](#footnote-ref-4)
5. https://en.wikipedia.org/wiki/SHA-2 [↑](#footnote-ref-5)
6. https://en.wikipedia.org/wiki/Hybrid\_cryptosystem [↑](#footnote-ref-6)
7. https://en.wikipedia.org/wiki/Advanced\_Encryption\_Standard [↑](#footnote-ref-7)
8. https://en.wikipedia.org/wiki/Public-key\_cryptography [↑](#footnote-ref-8)
9. https://aspe.hhs.gov/report/studies-welfare-populations-data-collection-and-research-issues/two-methods-linking-probabilistic-and-deterministic-record-linkage-methods [↑](#footnote-ref-9)
10. https://www.ncbi.nlm.nih.gov/books/NBK253312/ [↑](#footnote-ref-10)
11. https://www.cs.cmu.edu/~adamchik/21-127/lectures/graphs\_1\_print.pdf [↑](#footnote-ref-11)
12. https://www.sqlite.org/index.html [↑](#footnote-ref-12)
13. http://travistidwell.com/blog/2013/09/06/an-online-rsa-public-and-private-key-generator/ [↑](#footnote-ref-13)
14. http://www.hl7.eu/refactored/segPID.html [↑](#footnote-ref-14)
15. https://aspe.hhs.gov/report/studies-welfare-populations-data-collection-and-research-issues/two-methods-linking-probabilistic-and-deterministic-record-linkage-methods [↑](#footnote-ref-15)
16. https://github.com/sqlitebrowser/sqlitebrowser [↑](#footnote-ref-16)
17. https://en.wikipedia.org/wiki/SHA-2 [↑](#footnote-ref-17)