WASHINGTON STATE DEPARTMENT OF HEALTH

Record Deduplication: Reference Pipeline and Common Pitfalls

Jon Downs
Data Science and Engineering Unit Supervisor
Center for Health Statistics
DCHS

Washington State Department of HEALTH

DOH 422-287 May 2025

Table of Contents

Table of Contents

Introduction **3**

Example Pipeline: Record Deduplication **4**

Data Source **4**

Project Set-up **4**

Implementation **5**

Key Steps 5

Setup and Load Data 5

Blocking 7

Feature Engineering 9

Split into Training and Testing Datasets 10

Model Fit 11

Model Evaluation 12

Performance Considerations **14**

Avoid Rowwise Operations for Vectorized Functions **14**

Use the Right Function for the Job **16**

Use Functions as Intended **16**

Avoid multithreaded foreach on Windows **18**

Conclusion **20**

ACKNOWLEDGEMENTS: This work was possible due to the contributions and review of other Washington State Department of Health (DOH) Center for Health Statistics (CHS) staff. Maya Bhat, Sean Coffinger, and Julian Kapoor were the original authors of much of the code adapted herein. It is an honor to improve upon their valuable and important work. Sean Coffinger additionally provided feedback during the report writing process.

Introduction

Record linkage and record deduplication are a common tasks in vital records and public health. This is due to the nature of how data are received. The Center for Health Statistics (CHS) receives data from hospitals, primary care offices, medical examiners, funeral homes, and birthing centers. To facilitate this work, CHS has a dedicated Linkage and Integrated Data Analysis team, or LIDA. LIDA's primary focus is to prioritize equitable representation and identify links with high accuracy, but demands for better script runtimes and modularity have grown alongside the volume and scale of work. This is exacerbated by an anticipated loss of cloud computing power due to changes in the funding landscape. In 2025, LIDA partnered with the Data Science and Engineering team (DSE) at CHS to improve the performance and efficiency of some pipelines. This collaboration is ongoing, but many improvements have already been found.

LIDA previously created a demonstration version of a real production linkage pipeline based on their ECHIDNA linkage project. Others have since adapted this work for their own linkage processes. However, this pipeline is not optimized for performance, and it is not a reproducible example. A reproducible example is a combination of documentation, source code, and input data that allows someone to rerun a code example from start to finish and get exactly the same result. Reproducible examples are both a great resource for learning and proof that the code being provided works as described.

This document provides a performance-minded, reproducible guide to implementing a data deduplication pipeline. To start, we will walk through a reference data deduplication pipeline for a publicly available dataset. We use the term "reference pipeline" similarly to the concept of a golden test in computer science. The reference pipeline is a constant set of inputs where the truth is known or well-approximated. It serves two key purposes. First, testing a script against a golden test confirms consistent behavior. This is useful when migrating or upgrading software used to run a process. Second, it creates a controlled environment to test out new modeling strategies. A reference pipeline can be converted into production with minimal changes. Reference pipelines and other code testing strategies are highly encouraged for these reasons. After this, we highlight performance pitfalls discovered during a review of our own linkage pipelines. All of the examples in the performance pitfalls section are adapted from relevant sections of the ECHIDNA demonstration and evaluation project.

The intended audience for this document is intermediate R users with a background in public health or the social sciences. Contrary to other CHS linkage reports, the goal is not to achieve state of the art accuracy or optimal performance. Rather, it represents a base case framework that can be customized and optimized according to the needs of a project. We will highlight places where customization would likely improve the end product. Please refer to other CHS reports, such as the ECHIDNA project, for examples of fully customized linkage pipelines.

Example Pipeline: Record Deduplication

To begin, we will describe the basic steps of our pipeline. Given a table of records, the goal is to create a common person identifier to identify which records correspond to the same individual. There are many decisions to be made when designing a a deduplication pipeline, such as blocking methods and model choice. An overview of such decisions is beyond the scope of this article. There are many great overviews readily available, such as this white paper on blocking methods. CHS LIDA uses traditional blocking and supervised learning in most linkage pipelines. The primary models are support vector machines (SVM) and random forest (RF) models. Traditional blocking is a combination of exact and fuzzy matches on fields such as first name, last name, date of birth, and social security number. An exact match is one where the two fields are identical, and fuzzy matches allow some amount of pre-defined error between the two fields. For example, if you chose to block on exact DOB and a fuzzy match on last name, a John Doe born on 2/1/1980 would be in the same block as John Dove born 2/1/1980.

Our final product is a reference implementation. In a proper golden test, exact replication is expected. Here, we have a stochastic process with a model. Proving the whole system catches enough true links with few false positives is sufficient for now. In real projects, revisit this testing strategy once the process is mature.

Data Source

Data come from the RLdata10000 data source of the RecordLinkage R package. The dataset is artificially-generated records based on the most common names in Germany, and 10% of the records are duplicates that should link to another record in the dataset.

Project Set-up

All code was written in R version 4.3.2 in Windows on an x86-64 bit architecture. While it was not tested in other versions of R, we expect the code to run as expected on most modern versions of R. Users who wish to follow along with the code samples should clone the source code from the Github repo.

```
git clone https://github.com/DOH-JPD2303/LinkageRefImp
cd ./LinkageRefImp
```

The project includes a `renv` environment that can be used to recreate the exact package versions used. Those new to `renv` should read the Introduction to renv vignette. To get started with `renv`, run the following R code in the root of the project repo:

```
# Install renv, if necessary
# install.packages('renv')
renv::restore()
```

Alternatively, one could choose to install all necessary packages on their own:

```
install.packages(c(
        'data.table', 'RecordLinkage', 'stringdist', 'microbenchmark', 'snow',
        'foreach', 'doParallel', 'parallel', 'e1071', 'randomForest'
        ))
```

Implementation

Key Steps

Our reference pipeline contains the following steps:

1) Setup and Load Data
2) Blocking
3) Feature engineering
4) Split data into training and testing datasets
5) Model fit on training set
6) Model evaluation on testing set

One notable difference from a real-world, production pipeline is that the true and false links are already known. This is not a luxury available to any non-trivial linkage project. A real-world project might look more like this:

1) Setup and Load Data
2) Cleaning, standardizing and custom variable generation
3) Blocking
4) Feature engineering
5) Generate model predictions
   - If this is the first iteration, unsupervised methods or transfer learning will be required to generate the first set of links. This will likely require heuristics or a great deal of human review. The result of this work can then be used to train a custom supervised model.
6) If necessary, validate the model predictions to remove bad links, or catch links the model missed.
7) Assign a common person identifier to any successful links.
8) Retrain the linkage model as more links are made or the model stops improving.

Setup and Load Data

First, load all required packages and the target dataset.

```
# Load packages
library(data.table)
library(RecordLinkage)
library(e1071)
library(randomForest)
library(stringdist)
```

```
# Load data, convert to data.table
data(RLdata10000)
recs <- data.table(RLdata10000)
```

The `data.table` package offers a variety of performance improvements over the base `data.frame` R class. For more information, you might check out response to "why didn't you just enhance data.frame in R?" in the FAQ. Key `data.table` functions, such as joins and if-else statements, are parallelized under the hood (see package documentation for details). As of writing, the default `data.table` behavior is to use half of the logical cores available for parallelized tasks. It is a reasonable default: this should roughly match the total physical cores on most machines. Consider toggling the number of threads using the `setDTthreads` function if necessary.

Next, we perform some light data manipulation to facilitate blocking. First, we convert birth day/month/year to string variables. Next, we assign a row ID to uniquely identify each record. Finally, we assign a "person ID", which uniquely identifies persons, some of whom have 2+ records in the full dataset. In a production scenario, the goal would be to create and maintain the person ID variable.

```
# Universal type conversions, missingness handling, and assigning persistent IDs
recs[, `:=` (
        # Convert to string
        by = paste0(by),
        bd = paste0(bd),
        bm = paste0(bm),

        # Record and person identifiers
        row_id = .I,
        person_id = identity.RLdata10000,

        # Replace NA with blank strings
        fname_c1 = fcoalesce(fname_c1, ''),
        fname_c2 = fcoalesce(fname_c2, ''),
        lname_c1 = fcoalesce(lname_c1, ''),
        lname_c2 = fcoalesce(lname_c2, '')
)]
```

Next, let's create a dataset of true links to evaluate the performance of the rest of the pipeline. Following a join, the default `data.table` behavior is to add the prefix `i.` to any columns on the right-hand side if it shares a name with the left-hand side.

```
# Row pairs can come in a jumbled order
# This function subsets/deduplicates the dataset so only unique pairs remain
# The resulting df will include only row ID's, where the lowest ID in each
# pair is listed first.
row_id_dedup <- function(row_id, i.row_id) {
        df <- data.table('row_id' = row_id, 'i.row_id' = i.row_id)
        df[, `:=` (
                row_id = pmin(row_id, i.row_id),
```

```
                i.row_id = pmax(row_id, i.row_id)
        )]
        df <- unique(df)
        return(df)
}

# This is the source of truth: will come in useful in evaluation
truth <- recs[recs, on='person_id', nomatch=0][row_id != i.row_id]
truth <- row_id_dedup(truth$row_id, truth$i.row_id)
truth <- truth[recs, on='row_id', nomatch=0]
truth <- truth[recs, on=c('i.row_id' = 'row_id', 'person_id'), nomatch=0]
```

Blocking

Next we perform blocking. Without blocking, we would have to compare all records to one another. In our example dataset of 10,000 records, 49,995,000 unique comparisons are possible. With 1,000 true links, this means there are 49,994 bad candidates for each true link. Thus, for every true link there are enough bad candidates to sell out the next Seattle Mariners game, with 2,065 people left over. This is clearly a waste of computing resources. Blocking uses relatively cheap comparisons early in the linkage process to filter out unlikely matches while preserving as many likely candidates as possible. CHS pipelines tend to use traditional blocking based on exact and fuzzy string matches. For our reference implementation, we chose the following rules for blocking:

1) Exact match on year of birth, first name differs by no more than two characters, last name differs by no more than two characters.
2) Exact match on first name, last name differs by no more than two characters
3) Exact match on last name, first name differs by no more than two characters

Any pairs not meeting one or more of these criteria will be excluded from further analysis.

Blocking rules should be a key focal point of your design. Consider all available fields as well as the completeness/quality of those fields when determining blocking criteria. In our example, we have few fields and high data quality. If we had access to gender or social security numbers, those would be great things to block on. The hallmark of a good blocking algorithm is excluding as many true negatives as possible from further testing while maximizing true positives retained.

Here is the blocking code in the reference implementation:

```
# Exact on birth year
join_cols = c('by')
dob = recs[recs, on=join_cols, allow.cartesian=TRUE][row_id != i.row_id]
dob[, `:=` (
        fname_diff = stringdist(fname_c1, i.fname_c1, method='lv'),
        lname_diff = stringdist(lname_c1, i.lname_c1, method='lv')
)]
dob <- dob[fname_diff <= 2 | lname_diff <= 2]

# Exact on first name
```

```
join_cols = c('fname_c1', 'fname_c2')
first = recs[recs, on=join_cols, allow.cartesian=TRUE][row_id != i.row_id]
first <- first[stringdist(lname_c1, i.lname_c1, method='lv') <= 2]

# Exact on last name
join_cols = c('lname_c1', 'lname_c2')
last = recs[recs, on=join_cols, allow.cartesian=TRUE][row_id != i.row_id]
last <- last[stringdist(fname_c1, i.fname_c1, method='lv') <= 2]

# Join all candidates to a single list and deduplicate
keep_cols <- c('row_id', 'i.row_id')
candidates <- rbindlist(list(
        dob[, ..keep_cols], last[, ..keep_cols], first[, ..keep_cols]
))
candidates <- row_id_dedup(candidates$row_id, candidates$i.row_id)

# Add the other columns for feature engineering
candidates <- candidates[recs, on='row_id', nomatch=0]
candidates <- candidates[recs, on=c('i.row_id' = 'row_id'), nomatch=0]

# We have labeled data- use it to apply a yes/no indicator.
# This is our model target
candidates[, is_match := as.factor(person_id == i.person_id)]
```

Since we have a source of truth, let's see how the blocking algorithm did.

```
print_blocking_performance <- function(df, candidates, truth) {
        # Identify cases where truth and candidates agree
        agree <- candidates[truth, on = c('row_id', 'i.row_id'), nomatch=0]
        pct_agree <- formatC(100 * nrow(agree) / nrow(truth), format="f", digits
= 2)
        msg <- paste0(
                "Number of true matches found: ",
                nrow(agree),
                " (", pct_agree, "%)\n"
        )
        cat(msg)

        # How much have we reduced the search space?
        num_possible_pairs <- choose(nrow(df), 2)
        reduction_ratio <- 100 * (
                num_possible_pairs - nrow(candidates)
        ) / num_possible_pairs
        reduction_ratio <- formatC(reduction_ratio, format="f", digits = 2)
        msg <- paste0("Reduction ratio: (", reduction_ratio, "%)\n")
        cat(msg)
}

print_blocking_performance(recs, candidates, truth)
```

```
Number of true matches found: 997 (99.70%)
Reduction ratio: (99.89%)
```

Blocking reduced the number of records to compare by 99.89%, but 0.3% of true matches were also filtered out. Now, we only have around 34 candidates for every true match. Depending on your linkage aims and project objective, this may be suitable. If not, more flexible blocking parameters are warranted. Here, there is certainly room for improvement: we leave that exercise to the reader. Please see the ECHIDNA demonstration project or other CHS reports for some blocking strategies we have used in real projects.

Feature Engineering

Any model that compares words must numerically represent the difference between the two items being compared. In traditional models, we use string distance metrics. Here, we stick to a limited set of string comparisons. In a real pipeline, this is another place where customizations are encouraged.

```
column_pairs <- list(
        # LHS first name, first part
        c("fname_c1", "i.fname_c1"),
        c("fname_c1", "i.fname_c2"),
        c("fname_c1", "i.lname_c1"),
        c("fname_c1", "i.lname_c2"),

        # LHS first name, second part
        c("fname_c2", "i.fname_c1"),
        c("fname_c2", "i.fname_c2"),
        c("fname_c2", "i.lname_c1"),
        c("fname_c2", "i.lname_c2"),

        # LHS last name, first part
        c("lname_c1", "i.fname_c1"),
        c("lname_c1", "i.fname_c2"),
        c("lname_c1", "i.lname_c1"),
        c("lname_c1", "i.lname_c2"),

        # LHS last name, second part
        c("lname_c2", "i.fname_c1"),
        c("lname_c2", "i.fname_c2"),
        c("lname_c2", "i.lname_c1"),
        c("lname_c2", "i.lname_c2"),

        # Birth date (3 parts)
        c("by", "i.by"),
        c("bm", "i.bm"),
        c("bd", "i.bd")
)
```

```
# Apply Jaro-Winkler similarity to each pair
jw_names <- paste0(
        "JW_", sapply(column_pairs, function(x) paste0(x[1], "_", x[2]))
)
candidates[, (jw_names) := lapply(
        column_pairs,
        function(cols) stringdist::stringdist(get(cols[1]), get(cols[2]), method=
'jw')
)]
```

Here are a couple of potential features we might consider were we to improve upon this:

- Add another type of commonly utilized string distance, such as cosine distances
- Add indicators for when a string was empty or had very few characters
- Add handling for string distances when one string is very short
- Take the minimum of multiple string distances. This is useful if, say, it is common for a last name to wind up in the first name field.

Just remember parsimony is a virtue. It makes the pipeline more stable and reduces compute time. Start with a relatively simple base and use a reference dataset to prove that adding complexity improves the overall model fit.

Split into Training and Testing Datasets

Before training a model, we hold out 20% of all candidates as a test set. Models perform best on the data used to train them. The 20% of records held out will test how the model performs on data not used in training. This is a better approximation of how our model would behave in production.

```
# Set seed for reproducibility
set.seed(123)

# Controls percentage of records in training set vs. testing
train_ratio <- 0.80

# Create training and testing indices
train_indices <- sample(
        seq_len(nrow(candidates)), size = train_ratio * nrow(candidates)
)
train <- candidates[train_indices]
test <- candidates[-train_indices]

# Extract model features and outputs from each set
X_train <- train[, ..jw_names]
Y_train <- train$is_match
X_test <- test[, ..jw_names]
Y_test <- test$is_match
```

Model Fit

We will use an ensemble modeling approach in this implementation. The structure is as follows:

- Component models (feed into meta model)
    - A random forest model
    - A support vector machine
- Meta model: logistic regression

The component models each take all the string distances calculated in the 'feature engineering' section. The meta model takes the probabilities from the component models and uses them to produce a final determination. Especially for larger pipelines, you will not necessarily want to train a new model at each run of your pipeline. Thus, we save our model fit to file and load it for subsequent runs. Renaming or deleting the model weights file triggers a new round of model training.

```
# Output file for SVM model
svm_mod_fn <- file.path('./models/svm_mod.RDS')

# Make model directory if it does not exist
if(!dir.exists(dirname(svm_mod_fn))) {
        dir.create(dirname(svm_mod_fn))
}

# SVM model - train if needed, otherwise load pre-existing
svm_mod_fn <- file.path('./models/svm_mod.RDS')
if(!file.exists(svm_mod_fn)) {
        svm_mod <- svm(y = Y_train, x = X_train, probability=TRUE)
        saveRDS(svm_mod, svm_mod_fn)
} else {
        svm_mod <- readRDS(svm_mod_fn)
}

# RF model - train if needed, load if not
rf_mod_fn <- file.path('./models/rf_mod.RDS')
if(!file.exists(rf_mod_fn)) {
        rf_mod <- randomForest(y = Y_train, x = X_train)
        saveRDS(rf_mod, rf_mod_fn)
} else {
        rf_mod <- readRDS(rf_mod_fn)
}

# Meta model - train or load
meta_mod_fn <- file.path('./models/meta_model.RDS')
if(!file.exists(meta_mod_fn)) {
        # Get training set predictions to fit the meta model
```

```
        svm_train_preds <- predict(svm_mod, X_train, probability=TRUE)
        svm_train_probs <- attr(svm_train_preds, "probabilities")
        rf_train_preds <- predict(rf_mod, X_train, type="prob")

        # Input data for meta model
        metadata <- data.table(
                svm_t = svm_train_probs[, 2],
                rf_t = rf_train_preds[, 2],
                is_match = Y_train
        )

        # Fit model and save
        meta_model <- glm(is_match ~ ., data = metadata, family = "binomial")
        saveRDS(meta_model, meta_mod_fn)
} else {
        meta_model <- readRDS(meta_mod_fn)
}
```

Model Evaluation

Now we evaluate the model against the testing set.

```
# Custom function to run the whole inference pipeline
inference <- function(X, svm_mod, rf_mod, meta_mod) {
        svm_train_preds <- predict(svm_mod, X, probability=TRUE)
        svm_train_probs <- attr(svm_train_preds, "probabilities")
        rf_train_preds <- predict(rf_mod, X, type="prob")
        metadata <- data.table(
                svm_t = svm_train_probs[, 2],
                rf_t = rf_train_preds[, 2]
        )
        return(predict(meta_model, metadata, type="response"))
}

# Gets confusion matrix and evaluation stats
compute_metrics <- function(predicted, actual, positive_class) {
        # Convert predictions and actual labels to factors for consistency
        predicted <- factor(predicted, levels = levels(actual))
        actual <- factor(actual, levels = levels(actual))

        # Confusion matrix
        confmat <- table(Predicted = predicted, Actual = actual)

        # Extract true/false positives and negatives
        TP <- confmat[positive_class, positive_class]
        FP <- sum(confmat[positive_class, ]) - TP
        FN <- sum(confmat[, positive_class]) - TP
```

```
        TN <- sum(confmat) - TP - FP - FN

        # Output metrics
        PPV <- TP / (TP + FP)
        NPV <- TN / (TN + FN)
        Sensitivity <- TP / (TP + FN)
        Specificity <- TN / (TN + FP)
        F1 <- 2 * TP / (2 * TP + FP + FN)

        # Return results as a list
        return(list(
                Confusion = confmat,
                PPV = PPV,
                Sensitivity = Sensitivity,
                Specificity = Specificity,
                F1 = F1
        ))
}

# Get predictions on test set and evaluate
meta_preds <- inference(X_test, svm_mod, rf_mod, meta_model)
meta_metrics <- compute_metrics(as.factor(meta_preds > 0.5), Y_test, "TRUE")
print(meta_metrics)
```

```
$Confusion
         Actual
Predicted FALSE  TRUE
    FALSE 10995     5
    TRUE      5   177

$PPV
[1] 0.9725275

$Sensitivity
[1] 0.9725275

$Specificity
[1] 0.9995455

$F1
[1] 0.9725275
```

This pipeline captures 97.3% of true links. After considering the 0.3% of true links we lost in blocking, our overall process should successfully identify 97.0% of true links. This is a decent starting point, but could be improved further. One of the motivating principles for the LIDA team's work is that the 2-3% of links that standard tools systematically miss come from underserved groups. For a more complete accounting of the techniques LIDA uses to do this, please see their explainer of the ECHIDNA project.

Performance Considerations

Next, we discuss performance considerations. Before proceeding, a reminder: this paper is designed for those with a public health or social science background with intermediate R experience. Computer science has terminology and theory that are beyond the expertise of this article's authors. Curious readers should read about time complexity in computer science on their own. A good place to start is big O notation.

The first thing to understand is that languages like R and Python were designed for ease of use, not runtime performance. In scenarios where performance matters, users must rely on R/Python libraries written in faster languages such as C/C++. The key to optimizing R/Python code is to efficiently and correctly use these libraries. Secondly, the best way to increase performance is to do less stuff. The first priority is to identify and remove unnecessary and duplicative steps. This typically enhances the readability of the code, as well. After a point, further optimization likely means more complex code.

With that, let's discuss some performance wins found in our review of CHS pipelines. Some of the examples below require the creation of new variables in our `candidates` dataset from above:

```
candidates[, `:=` (
        DOB = as.Date(paste(by, bm, bd, sep="-"), format="%Y-%m-%d"),
        i.DOB = as.Date(paste(i.by, i.bm, i.bd, sep="-"), format="%Y-%m-%d")
)]
```

We also create a "big" version of the dataset to see how our code snippets might perform on a larger set of data:

```
big_candidates <- rbindlist(rep(list(candidates), 10))
```

Avoid Rowwise Operations for Vectorized Functions

Many performance-optimized R functions are vectorized, meaning they take a vector as arguments. This is most effective when the underlying function is written in a faster language such as C. This way, C can do as much work as possible before incurring the overhead of sending the object back to the R session. This is why R users are told that for loops are slow. A corollary to this is that rowwise operations are slow. Take this example of a code snippet adapted from the ECHIDNA demo:

```
candidates[, 'DOB_HAM' := stringdist(DOB, i.DOB, method="hamming"), by=.I]
```

The `by=.I` turns this into a rowwise operation. The ECHIDNA demo includes the following note above this block of code (edited for clarity):

> *"the '[by=.I]' grouping variable...tells DT to work in a row-wise fashion. In simple calculations...it is unnecessary but doesn't slow down DT at all."*

This is not entirely correct. A fully vectorized version of this function produces an identical result in a fraction of the time:

```
timing <- microbenchmark(
        bygroup = candidates[, 'DOB_HAM' := stringdist(DOB, i.DOB, method="hammin
g"), by=.I],
        noby = candidates[, 'DOB_HAM2' := stringdist(DOB, i.DOB, method="hamming"
)],
        times = 5
        )
print(timing)

Unit: milliseconds
    expr       min        lq       mean    median        uq       max neval
 bygroup 1782.3194 1797.5544 1816.89428 1806.0168 1806.6876 1891.8932     5
    noby   45.2718   46.0695   47.76328   46.4201   49.9911   51.0639     5

print(identical(candidates$DOB_HAM, candidates$DOB_HAM2))

[1] TRUE
```

The intended message of the note in the ECHIDNA tutorial was that the performance difference was ignorably small. At large enough scale, the difference becomes unignorable, however:

```
timing <- microbenchmark(
        bygroup = big_candidates[, 'DOB_HAM' := stringdist(DOB, i.DOB, method="ha
mming"), by=.I],
        noby = big_candidates[, 'DOB_HAM2' := stringdist(DOB, i.DOB, method="hamm
ing")],
        times = 5
        )
print(timing)

Unit: milliseconds
    expr        min         lq       mean    median         uq        max neval
 bygroup 18265.4656 18447.5255 18638.6676 18629.585 18825.2686 19020.9519     3
    noby   445.3976   446.5203   451.6126   447.643   454.7201   461.7971     3
```

With 10x the rows to process, each method took 10x longer to run. So it appears to be scaling linearly. When milliseconds become minutes, a 40x difference in runtime matters. There are two reasons why the rowwise function performs so slowly. First, rather than calling a C function once with many input pairs, rowwise operations force a C call for each row in the dataset. The overhead of converting from C to R is incurred many, many times. Second, `stringdist` is both vectorized and multithreaded. If given two vectors, it will spread the work across multiple processor cores to speed things up. But the rowwise operation forces `stringdist` to consider each row individually.

Use the Right Function for the Job

Later on in the ECHIDNA demo, the `max` and `min` are combined with a rowwise operation to get the rowwise maximum/minimum values. But there is built-in R function for this, `pmax`:

```
print(microbenchmark(
        rowwise = candidates[, MAX_DOB := max(DOB, i.DOB), by=.I],
        vectorized = candidates[, MAX_DOB2 := pmax(DOB, i.DOB)]
        ))

Unit: milliseconds
       expr      min       lq      mean   median       uq      max neval
    rowwise 138.1928 138.6273 139.83654 138.6489 140.8553 142.8584     5
 vectorized   1.3857   1.6274   1.71172   1.6818   1.9126   1.9511     5

print(identical(candidates$MAX_DOB, candidates$MAX_DOB2))

[1] TRUE
```

There is no substitute for reading the documentation for frequently used packages and functions. You will be surprised how much you will learn!

Use Functions as Intended

We also found issues in the branching logic used within our string distance calculations. As mentioned earlier, we often want to modify q-gram based metrics when one string has fewer characters than q. Here is an adapted example from the ECHIDNA paper:

```
candidates[, 'FIRSTNAME_COS' := stringdist(
        fname_c1,
        i.fname_c1,
        method = c('cosine'),
        q = ifelse(
                nchar(fname_c1) < 3 | nchar(i.fname_c1) < 3,
                min(nchar(fname_c1), nchar(i.fname_c1)),
                3
                )
), by=.I]
```

There are three problems with the this code. First, we are doing an avoidable rowwise operation. Second, `data.table` has an `fifelse` function that is multithreaded and much faster than the base R implementation (read the docs!). Third, q is meant to take only a single integer, but instead a vector of values is provided. The third problem does not affect performance, but it does create an incorrect result. This can be seen by comparing it to an alternative approach (that is also 100x faster):

```
# Cosine edit distance with flexible handling of small strings
cosine_qflex <- function(left, right, max_q=3) {
        # Get minimum length between each pair
        min_length <- pmin(nchar(left), nchar(right))

        # Start output vector
        out <- vector("numeric", length = length(left))

        # Populate each case
        q_seq <- seq(max_q)
        for(q in seq_along(q_seq)) {
                # For all integers up to the last, we only check for equality.
                # In the last iteration, we include everything else
                if (q == length(q_seq)) {
                        idx <- which(min_length >= q)
                } else {
                        idx <- which(min_length == q)
                }

                # Run the string distance function
                out[idx] <- stringdist::stringdist(
                        left[idx], right[idx], method = 'cosine', q = q
                )
            }
        return(out)
}

candidates[, FIRSTNAME_COS_C2_ALT := cosine_qflex(fname_c2, i.fname_c2)]

# In this case, results are different. Preview the first few instances.
examine_cols <- c('fname_c2', 'i.fname_c2', 'FIRSTNAME_COS_C2', 'FIRSTNAME_COS_C2
_ALT')
print(head(
        candidates[FIRSTNAME_COS_C2_ALT != FIRSTNAME_COS_C2, ..examine_cols]
))

   fname_c2 i.fname_c2 FIRSTNAME_COS_C2 FIRSTNAME_COS_C2_ALT
     <char>     <char>            <num>                <num>
1:   WERNER     WERNIR        0.0000000                  0.5
2:   DIETER    JUERGEN        0.0000000                  1.0
3:     OLAF       OLAF        0.3291796                  0.0
4:     OLAF      HEINZ        0.0000000                  1.0
5:   VOLKER    JOACHIM        0.0000000                  1.0
6:    KLAUS    JOACHIM        0.0000000                  1.0
```

The exact behavior behind this bug has been hard to pin down, but it has to do with how q is defined. When multiple arguments are passed to q, it ignores all but the first argument. This can be made clear with a simpler example:

```
a <- c('apple', 'banana', 'cavendish')
b <- c('aple', 'bananas', 'Kaepernick')

# Note both objects have q = 1 for the first-listed q argument
out1 <- stringdist::stringdist(a, b, q=1, method='cosine')
out2 <- stringdist::stringdist(a, b, q=c(1, 1056, 10e7), method='cosine')

print(out1 == out2)
# TRUE TRUE TRUE
```

Try not to get too clever, and always test your code. In general, avoid branching logic within a function to assign a variable that's not meant to accept a vector of arguments.

Avoid multithreaded foreach on Windows

Many of our pipelines were using the `foreach` and `doParallel` packages in R. After setting up parallelization, we split data into equal chunks, one for each process:

```
library(snow)
library(foreach)
library(doParallel)
library(parallel)

# Set up parallel processes
no_cores <- 6
cl <- makeCluster(no_cores, type="PSOCK")
registerDoParallel(cl)

# Make a sufficiently large dataset for demonstration
example <- rbindlist(rep(list(candidates), 20))

# Split data into one group for each process
example[, grp := cut(seq_len(nrow(example)), breaks=no_cores, labels=FALSE)]
example <- split(example, by='grp')
```

Then, each chunk is processed on a worker thread using `foreach` and `doParallel`. The code snippet is wrapped in `system.time()` to track total time. Note that `microbenchmark` is not used as it can be misleading for user-defined parallelism.

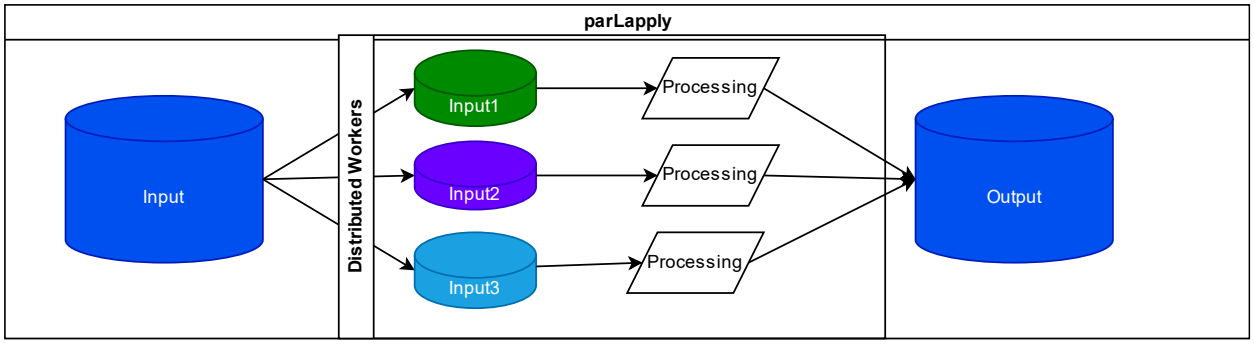
```
foreach_time <- system.time({
        # For-each, works best forking, which is not available on Windows
         foreach_result <- foreach(
                i = 1:no_cores,
                .packages = c('data.table', 'stringdist'),
                .export="example",
```

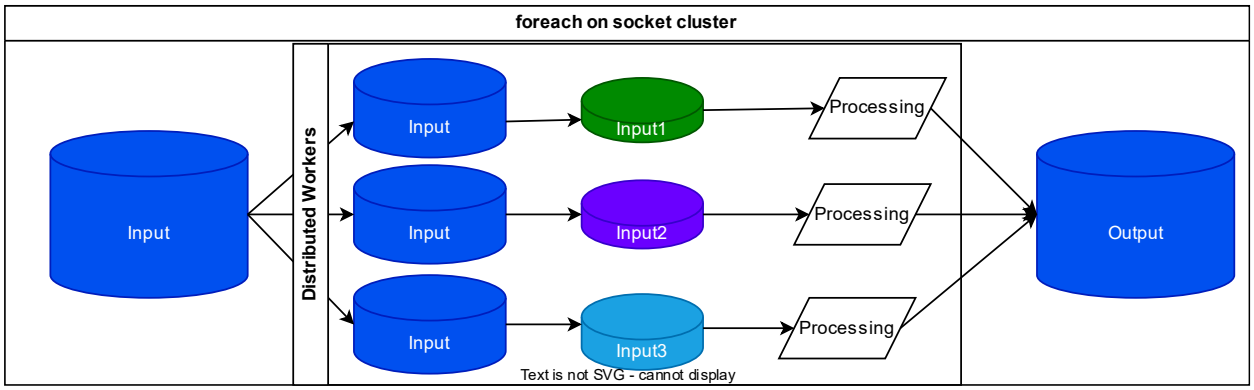
```
                .combine=c
        ) %dopar% {
                x <- example[[i]]
                output <- stringdist::stringdist(x$by, x$i.by, method='jw', nthre
ad=1)
                return(output)
        }
})
print(foreach_time)

   user  system elapsed
   2.12    6.04    9.70
```

How about we compare this to something using `parLapply`?

```
parlapply_time <- system.time({
        # Performs better on Windows
        parlap_result <- do.call(c, parLapply(
                cl, example,
                function(x) stringdist::stringdist(x$by, x$i.by, method='jw', nth
read=1)
        ))
})
print(parlapply_time)

   user  system elapsed
   0.36    1.11    1.70
```

In this case, `parLapply` is about 5.7x faster. This is because Windows does not allow process forking, a concept from Unix-based systems such as MacOS and Linux. Instead, parallelism must be implemented through sockets, which requires each worker to have its own memory space. The `foreach` and `doParallel` packages are designed for forking. In socket clusters, they copy memory over to each worker. As implemented, much of the memory copied over is unnecessary. Using `parLapply` addresses this by sending each worker process only the data it is responsible for. Each process is depicted graphically below. The timing outputs show that the `foreach` loop spent nearly 5 seconds more time in system calls than `parLapply`. Note that memory allocation is a system call. Curious readers can learn more about the different kinds of parallelism in R here and here.





Conclusion

In this paper, we have implemented a fully reproducible record deduplication pipeline and provided some general tips for improving performance. This effort was informed by our own efforts to standardize and optimize our own existing processes. We hope the reader finds this useful in their own processes. Future papers will build off this work to implement more advanced functionality, such as writing custom Rust code to improve the memory efficiency of fuzzy joins.

DOH 422-287 May 2025

To request this document in another format, call 1-800-525-0127. Deaf or hard of hearing customers, please call 711 (Washington Relay) or email doh.information@doh.wa.gov.