Entity Deduplication for Veridion Data Engineer Internship

A Detailed, Large-Scale Approach to Noisy Company Records

(Bîrsan Gheorghe-Daniel)

Contents

1 Introduction

1.1 Context and Rationale

In modern data environments, companies often receive large, *noisy datasets* from multiple sources, each using different standards for naming, location, industry codes, and so on. The **task** here is to **identify unique companies** among these messy records, gather duplicates into clusters, and produce a *single enriched row* for each cluster. This challenge was presented specifically for a *Veridion Data Engineer Internship* assignment but is relevant to many data integration scenarios.

1.2 Motivation

- Data duplication leads to inflated statistics, confusion, and wasted resources.
- Combining partial or inconsistent data into the "best" or "golden" record yields the most robust dataset.
- Speed and correctness are both crucial: blocking or indexing ensures we do not blow up computationally.

1.3 Overall Two-Part Pipeline

In broad terms, the approach can be split into two phases:

1. **Normalization & Preprocessing**:
   - Cleaning textual columns, normalizing websites, approximating missing location fields, merging or inferring `company_name` from other columns.
   - Producing a `df_cleaned` that is consistent and more easily comparable.
2. **Similarity & Clustering**:
   - Using a blocking strategy (e.g., `main_country`) to keep it fast.
   - A scoring system (points-based or fuzzy approach) to decide if pairs are duplicates.
   - Converting these duplicates into connected components and merging each cluster into a single row.

2 Challenge & Constraints

2.1 The Task

We are given a Parquet file with company records. The columns span:

- **Company Identifiers:** `company_name, company_legal_names, ...`
- **Location Data:** `main_country_code, main_region, main_city, ...`
- **Contact:** `primary_phone, phone_numbers, ...`
- **Web Presence:** `website_url, website_domain, facebook_url, ...`
- **Business Details:** `company_type, year_founded, ...`
- **Industry Codes:** `naics_vertical, main_business_category, ...`
- **Other Data:** `revenue, employee_count, inbound_links_count, ...`

The dataset is known to be:

- *Noisy* (missing, partial, or contradictory fields).
- *Inconsistent* (some records only have `company_name`, others only have a partial `company_legal_names`, etc.).
- Potentially *large*, requiring efficient grouping of duplicates.

2.2 Goals & Requirements

1. **Speed**: Must handle tens (or hundreds, I hope not :*(but it is quite fast) of thousands of rows quickly.
2. **Correctness**: Identify duplicates with minimal false positives or false negatives.
3. **Robustness**: Should handle unexpected or missing data gracefully.
4. **Development Feasibility**: Code should be maintainable, understandable, and easily adapted for future improvements.

3 Solution Outline

The final approach divides the solution into **three major parts**:

3.1 Part 1: Data Normalization and Preprocessing

- **Enrich Company Name:** If `company_name` is missing, use `company_legal_names`, or `short_description`, or parse the `website_domain`.
- **Location Approximation:** If `main_country` is missing, parse from `main_country_code` or from TLD. We skip advanced geocoding to remain offline and fast.

- **Activity Sector Normalization:** Combine codes like `naics_2022_primary_code`, `sic_codes`, or parse textual columns with an ML model.
- **Numeric Columns:** Convert `revenue`, `employee_count` to numeric if possible, ignoring invalid strings.

3.2 Part 2: Blocking and Finding Duplicates

- **Blocking by Country:** Only compare records within the same country. If `main_country` is missing, group those in a *"missing"* or *"unknown"* block.
- **Scoring System:** Assign points (or partial fuzzy scores) for matching columns:
  - **Company Name** (fuzzy ratio via RapidFuzz).
  - **Website Domain** (exact match).
  - **Social Media** links (partial or exact match).
  - **Revenue** or **Employee Count** if within $\pm 10\%$.
- **Thresholding**: If the total points exceed (e.g.) `4`, mark them as duplicates.

3.3 Part 3: Clustering & Enriching the Data

- **Connected Components**: Build a graph where each record is a node, each duplicate pair is an edge. All records in the same connected component are duplicates of each other.
- **Merging Rows**: For each cluster, pick the best data:
  - *Longest textual field* (ensures maximum detail).
  - *Largest numeric* or *average*, depending on domain preference.
  - *First non-null* fallback if no others exist.
- **Save Output**: The final DataFrame of deduplicated rows is stored in `final_data.parquet`.

4 Problems Encountered

Despite the plan, we ran into:

4.1 Naive Approach with Column-by-Column Compare

A first naive approach attempted to compare every row with every other row and check all columns if not null. This took **13 hours** and was obviously too slow.

4.2 Overly Detailed Location Enrichment

We tried geocoding via external APIs, but:

- Incurred network latency.
- Data usage / external calls overhead.
- Not feasible for a purely offline, time-sensitive solution.

Hence, we pivoted to TLD-based or code-based approximations.

4.3 General Speed vs. Coverage Trade-off

While more advanced or multi-level blocking might increase coverage,I had to ensure I didn't lose reliability. Ultimately, a single country-based block plus an optional fallback for missing countries was sufficient. I also tried web scrapping, to ensure the data but it was a time consuming running process so I gived up, because of hardware and time constraints. Also, it was possible to enrichmore the activity codes, but I didn't succeded so I tried to focus on other criterias giving the development time constraint

5 Detailed Documentation of the Code

5.1 Data Normalization (Step 1)

5.1.1 Logic Explanation

- `best_company_name(row)` obtains the best possible name by:
  1. Checking `company_name` or fallback to `company_legal_names`.
  2. If missing, parse `short_description` or `generated_description`.
  3. If still missing, parse `website_domain` (remove suffixes like "inc" or "ltd").
- `fill_missing_location(row)` tries to approximate `main_country` from `main_country_code` or from the TLD of `website_url`.
- `unify_activity_info(row)` merges multiple industry codes and short descriptions into a single `activity_enriched` field.

Illustration Code (Excerpt):

```
def best_company_name(row: dict) -> str:
    web = row.get("website_url")
    if pd.isna(web):
        web = None

    if web:
        d, _ = parse_domain_tld(web)
        if d:
            return remove_company_suffixes(normalize_text(d))

    # Fallback logic with short_description, etc.
    for desc_col in ["short_description", "long_description", "generated_description"]:
        val = row.get(desc_col)
```

```python
        if val and isinstance(val, str):
            # use first n words
            first3 = get_first_n_words(val, 3)
            return remove_company_suffixes(normalize_text(first3)) or None

    # final fallback: company_name or commercial_names
    ...
    return None
```

5.1.2 Data Example

Imagine a row with:

- `website_url` = `http://www.testcompany.com`
- `company_name` = `NA`
- `short_description` = `“leading test solutions”`

I parse out “testcompany” from `website_url` if no better name is found.

5.2 Finding Duplicates (Step 2)

5.2.1 Short Description of the Approach

We block by `main_country`. So it do not compare records from the US with records from Germany (unless the country is missing). We define a function `compute_pair_score(rowA, rowB)` awarding points for:

- **Name similarity** (via `rapidfuzz.fuzz.ratio`).
- **Exact domain match** (some columns are strictly 1 point or 2 points if identical).
- **Revenue or Employee Count** within $\pm 10\%$.
- **Social media** links if exact matches.

If total points exceed a threshold (like 4), we mark them as a likely duplicate pair.

5.2.2 Scoring System & Code Explanation

```python
def within_10pct(valA, valB):
    if valA is None or valB is None:
        return False
    try:
        vA = float(valA); vB = float(valB)
    except:
        return False
    if vA == 0 or vB == 0:
        return False
    ratio = vA / vB
    return (0.9 <= ratio <= 1.1)

def compute_pair_score(rowA, rowB):
    points = 0
    # Name fuzzy => up to 2 points
```

```
    ratio = fuzz.ratio(rowA["company_name"] or "", rowB["company_name"] or "")
    if ratio >= 80: points += 2
    elif ratio >= 50: points += 1

    # Website domain exact => +2
    if rowA["website_domain"] and rowA["website_domain"] == rowB["website_domain"]:
        points += 2

    # Revenue or employee_count within +/- 10% => +1 each
    if within_10pct(rowA["revenue"], rowB["revenue"]):
        points += 1
    if within_10pct(rowA["employee_count"], rowB["employee_count"]):
        points += 1

    # Bonus for social links
    # ...
    # Punishment for different cities/regions
    # ...
    return points
```

After computing these pairwise scores, we store pairs with `score >= 4` in `df_duplicates`.

5.3 Clustering and Enriching Data (Step 3)

5.3.1 Short Description of the Approach

We treat each record as a graph node. If two records are duplicates, we add an edge. Then **all connected nodes** belong to the same cluster. We unify them with a function that picks the best column value:

- *Longest text* for textual columns.
- *Largest* or *first non-null* for numeric columns.

5.3.2 Merging Code (Detailed)

```
import networkx as nx

G = nx.Graph()
for row in df_duplicates.itertuples():
    idxA = row.idxA
    idxB = row.idxB
    G.add_edge(idxA, idxB)

components = list(nx.connected_components(G))

def merge_cluster(cluster_indices, df):
    subset = df.loc[list(cluster_indices)]
    merged_row = {}
    for col in df.columns:
        colvals = subset[col].dropna().tolist()
        if len(colvals) == 0:
            merged_row[col] = None
            continue
        # if text, pick longest
        # if numeric, pick largest, etc.
    return merged_row
```

We then iterate through each connected component, *merge* those rows, and produce a final DataFrame. Finally, we write that DataFrame to `final_data.parquet`.

6 Analysis of Criteria

6.1 Correctness

- By carefully normalizing columns and awarding points for name, domain, location, etc., we reduce the chance of false positives.
- Using *connected components* ensures that if A matches B, and B matches C, then A, B, C are all considered duplicates.

6.2 Robustness

- Missing columns default to `None`, skipping or awarding 0 points.
- We do not rely on external APIs for location; thus no blocking network calls.
- If the dataset is partially large, we can scale with Dask or Spark if needed.

6.3 Code Quality

- Each transformation is in a clear function: `best_company_name`, `fill_missing_location`, ...
- The final merging code is also modular.
- Comments and docstrings are used for clarity.

6.4 Extra Mile

- We used **longest text** logic in merging to preserve maximum detail.
- We handled optional columns like social media and partial phone matches for partial scoring.
- We gave expansions for *revenue* or *employee_ count* within ±10%.

6.5 Presentation

We carefully outlined the approach in steps, each with code snippets and logic explanation. We also included a verification snippet that prints original cluster rows for a quick sanity check.

7 Future Adjustments

Even after building a stable approach, we foresee potential expansions:

- **LSH (Locality-Sensitive Hashing)**: If the name fields are extremely large or we approach millions of records, an LSH-based approach can drastically reduce pairwise comparisons.
- **Weighted Merging**:
  - Instead of simply "largest numeric" or "longest text," we can track *confidence* in each record or do a majority vote.
- **Language Model Summaries**: We might unify textual columns by concatenating or summarizing them with an ML model to produce the final text description.
- **Advanced *k*-Means or DBSCAN** Over Embeddings: Instead of pairwise scoring, embed the records in a vector space (like a *Transformer-based* embedding) and use clustering in high-dimensional space. This might be slower or more complex but could capture deeper semantic similarities.
- **Spark-based Graph Merging**: If the data is extremely large, we can use Apache Spark GraphX or GraphFrames for distributed connected-component detection.

8 Conclusion

1. **We split the problem** into a first step for *data normalization* and a second step for *finding and merging duplicates.*
2. **Normalization** ensures each column (e.g. `company_name`, `main_country`, `revenue`) is consistent, numeric fields are typed properly, and textual fields are trimmed.
3. **Similarity** uses blocking on country, a points-based system for columns, and a threshold for duplicates.
4. **Clustering** is done by turning duplicates into a *graph* and identifying *connected components.*
5. **Merging** finalizes the golden record, choosing the best textual or numeric values.

Thus, the final DataFrame in `final_data.parquet` is *fully deduplicated* and *enriched* with the best available data from each cluster of duplicates.

Why this Approach?

- It is *fast* enough for moderate to large data, especially with an offline, TLD-based location fill and a country-based block.
- It is *robust* in that we handle missing data gracefully.
- It is *easy to adjust* the scoring system or the merging logic as domain knowledge evolves.

9 Results

After executing the full deduplication pipeline on the cleaned dataset, the following outcomes were obtained:

- **Total Connected Components (Clusters):** 24,445
  - This means the dataset was segmented into 24,445 distinct groups of records.
  - A component with only one node represents a unique company without detected duplicates.
  - Components with more than one node indicate detected duplicates, grouped together.
- **Potential Duplicate Pairs Found (score $\geq$ 4):** 13,477
  - These represent candidate record pairs with high similarity based on the scoring system.
  - The score threshold of 4 was determined based on empirical testing to balance recall and precision.
  - The scoring system included fuzzy name match, website match, employee count similarity, revenue range, and social media overlap.

These results show that nearly a third of the dataset involved potential duplication, underlining the importance of this deduplication process. The graph-based clustering then ensured all linked duplicate pairs were unified into single golden records.

The final deduplicated and enriched dataset was saved as:

- **Filename:** `final_data.parquet`
- **Format:** Apache Parquet (columnar storage, optimized for large-scale analytics)

Final Words: The *Entity Deduplication* method proposed meets the requirements for a **Veridion Data Engineer Internship** by balancing speed, correctness, running time, and development effort. Our detailed code and approach demonstrate professional best practices in data normalization, fuzzy matching, cluster merging, and overall pipeline structure.