Structural Paradigms in Entity Resolution: A Comprehensive Study of Name Matching Methodologies in SQL and Python

The proliferation of distributed data architectures and the resultant fragmentation of identity across disparate systems have elevated entity resolution to a critical function within the enterprise data stack. Name matching, a primary component of entity resolution, involves the identification of records that refer to the same individual or organization despite variations in spelling, formatting, and cultural naming conventions. The technical landscape for addressing these challenges is bifurcated between Structured Query Language (SQL) environments, which offer high-performance data retrieval and governance, and the Python ecosystem, which provides the flexibility of advanced machine learning and complex text manipulation. This report evaluates the best-in-class approaches within both domains, analyzing the mathematical foundations of similarity metrics, the architectural trade-offs of various database engines, and the emergence of transformer-based semantic matching.

Foundations of String Similarity and Phonetic Heuristics

The mechanical comparison of name strings relies on two primary categories of algorithms: character-based distance metrics and phonetic encoding. Character-based metrics quantify the physical difference between two sequences of characters, while phonetic algorithms attempt to reconcile variations based on linguistic pronunciation.

Character-Based Edit Distances

The most enduring metric in this category is the Levenshtein distance, which calculates the minimum number of single-character operations—specifically insertions, deletions, or substitutions—required to transform one string into another. Mathematically, the Levenshtein distance between two strings a and b is defined as lev_{a,b}(|a|, |b|), where:
While the Levenshtein distance is highly effective for catching typographic errors and optical character recognition (OCR) failures, it does not account for transpositions, which are common in human data entry. To address this, the Damerau-Levenshtein distance incorporates transpositions of two adjacent characters as a single operation.
A more specialized metric for human names is the Jaro-Winkler similarity. The Jaro similarity d_j is calculated based on the number of matching characters m and the number of transpositions t :
The Winkler modification enhances this score by applying a prefix scale p, which grants higher similarity to strings that share a common prefix of length l up to four characters :
This prefix weighting is critical for name matching because human errors are statistically less

likely to occur in the initial characters of a surname or given name.

Phonetic Encoding and Multilingual Constraints

Phonetic algorithms map strings to codes based on their pronunciation in a target language. The Soundex algorithm, patented in 1918 and utilized extensively in the 1880–1910 United States Censuses, remains a foundational tool in most relational databases. Soundex reduces a name to a four-character code, preserving the first letter and mapping subsequent consonants to numbers while ignoring vowels and certain silent consonants. For example, the names "Smith" and "Smyth" both resolve to the code S530.

Algorithm	Mechanism	Primary Strength	Known Limitation
Soundex	Phonetic (English-biased)	High speed; low memory footprint	Limited to English; high false-positive rate
Metaphone	Phonetic (Rules-based)	Better phonetic precision than Soundex	Computationally more intensive than Soundex
Double Metaphone	Phonetic (Multilingual)	Handles non-Latin roots (Slavic, French)	Complex to implement in standard SQL
Levenshtein	Character Edit Distance	Language-agnostic; catches typos	Computationally expensive (O(n \cdot m))
Jaro-Winkler	Prefix-weighted similarity	Optimized for short names	Sensitive to string order and length

While Soundex is efficient for initial candidate generation, its reliance on English pronunciation makes it unsuitable for global datasets. The Double Metaphone algorithm improves upon this by generating primary and secondary codes to account for ambiguous pronunciations in diverse linguistic roots, such as "Schmidt" (primary: XMT, secondary: SMT) versus "Smith" (primary: SM0, secondary: XMT). However, for high-precision multilingual requirements, Beider-Morse Phonetic Matching is preferred, as it incorporates linguistic rules for Cyrillic, Hebrew, and Latin scripts.

High-Performance Name Matching in SQL Architectures

SQL remains the standard interface for enterprise analytics due to its ability to process billion-row datasets directly where the data resides, eliminating the overhead of data movement. Modern SQL engines have integrated fuzzy matching through specialized indexing and native functions.

PostgreSQL: Trigram Indexing and the fuzzystrmatch Extension

PostgreSQL provides a robust framework for fuzzy matching through the fuzzystrmatch and pg_trgm extensions. Trigram matching decomposes strings into three-character sequences (e.g., "apple" becomes {" a", " ap", "app", "ppl", "ple", "le "}) and calculates similarity based on the overlap of these sets.

A critical architectural decision in PostgreSQL involves the choice between Generalized Inverted Index (GIN) and Generalized Search Tree (GiST) indexes for trigram operations. GIN

indexes are typically best for static or read-heavy data because lookups are approximately three times faster than GiST. Conversely, GiST indexes are lossy—relying on hash-based signatures—but are faster to update and support "nearest-neighbor" searches using the <-> operator, allowing the database to return results in order of similarity without scanning the entire table.

SQL Server: Native 2025 Functions and CLR Implementations

For decades, Microsoft SQL Server developers relied on the SOUNDEX() and DIFFERENCE() functions for phonetic matching. However, these functions offered limited accuracy. Advanced fuzzy matching often required the implementation of Common Language Runtime (CLR) functions, where C# code was compiled and executed within the SQL process. Benchmarks indicate that CLR implementations of Levenshtein distance are often 30 times faster than T-SQL equivalents because T-SQL is an interpreted language optimized for set-based logic rather than iterative character processing.

In early 2025, Microsoft introduced native fuzzy matching functions in Azure SQL and SQL Database in Microsoft Fabric, including EDIT_DISTANCE, EDIT_DISTANCE_SIMILARITY, and JARO_WINKLER_SIMILARITY. These native functions provide the performance of compiled code with the simplicity of standard T-SQL, eliminating the security and maintenance overhead of CLR assemblies.

Google Cloud: BigQuery and Spanner Search Functions

Google Cloud's data platforms emphasize scalability for massive datasets. BigQuery provides LEVENSHTEIN_DISTANCE and SOUNDEX functions, but for high-volume searches, it leverages specialized search indexes. While the BigQuery SEARCH function is optimized for point lookups, more flexible matching often involves JavaScript User-Defined Functions (UDFs) to implement custom logic.

Google Spanner offers advanced full-text search capabilities through SEARCH_NGRAMS and SCORE_NGRAMS. These functions allow for approximate matching by identifying rows that share common n-grams with the search query. For example, "Kaliphorn" can match "California" based on shared trigrams like "ali" and "orn". Spanner also provides native SOUNDEX support for phonetic search.

Snowflake: Cortex AI and Vector Similarity

Snowflake has positioned itself at the forefront of AI-driven entity resolution through its Cortex AI suite. Unlike traditional databases that rely solely on lexical overlap, Snowflake's AI_SIMILARITY function computes similarity based on vector embeddings, enabling semantic matches that account for synonyms and conceptual relationships.

The Snowflake hybrid entity resolution workflow typically combines two paths:

1. **Fast Path (Vector Similarity):** Semantic embeddings are used to identify high-confidence matches with similarity \ge 80\%, bypassing expensive LLM calls for clear-cut cases.
2. **Smart Path (AI Classify):** For ambiguous cases where multiple candidates exist, Snowflake uses AI_CLASSIFY to intelligently select the best match based on the broader context of the record.

SQL Engine	Best-in-Class Approach	Key Functions	Indexing Strategy
PostgreSQL	Trigram Similarity	similarity(), levenshtein()	GIN (lookup) / GiST (ordering)
SQL Server	Native 2025 Functions	JARO_WINKLER_SIMILARITY	Columnstore / Standard B-Tree
Snowflake	Hybrid Cortex AI	AI_SIMILARITY, AI_EMBED	Vector Search Service
BigQuery	Search Indexes / UDFs	LEVENSHTEIN_DISTANCE	Managed Search Indexes
Spanner	N-Gram Search	SEARCH_NGRAMS, SOUNDEX	SEARCH INDEX on TOKENLIST

The Python Ecosystem: Versatility and Scalability

While SQL excels at data retrieval, Python is the preferred environment for complex data cleaning, machine learning integration, and custom business logic. The Python landscape for name matching is defined by high-performance string libraries and sophisticated record linkage frameworks.

Optimized Comparison Libraries: RapidFuzz vs. Jellyfish

In the Python ecosystem, processing speed is a critical differentiator. RapidFuzz, a library built with a C++ backend and SIMD (Single Instruction, Multiple Data) optimizations, has emerged as the best-in-class tool for string comparison. Benchmarks show that RapidFuzz is approximately 40% faster than older libraries like FuzzyWuzzy (now TheFuzz) while maintaining efficient memory usage.
Jellyfish remains a specialized tool for phonetic matching, supporting a wide range of algorithms including NYSIIS, Metaphone, and Match Rating Approach. However, researchers have noted that Jellyfish struggles with long text inputs compared to RapidFuzz.

Scalable Record Linkage with Splink

For large-scale deduplication, the Python Splink library is the industry standard. Splink utilizes the Fellegi-Sunter probabilistic model to estimate the likelihood that two records refer to the same entity. To overcome the O(n^2) complexity of pairwise comparisons, Splink leverages "blocking rules" to generate a subset of likely candidate pairs before applying detailed scoring. One of Splink's primary advantages is its performance; it uses DuckDB as its default backend for fast parallelized execution on a single machine, or Spark for distributed environments. Benchmarks demonstrate that Splink can deduplicate a 7-million-record dataset in just over two minutes at a minimal cloud cost. The library's "Comparison Library" allows for highly nuanced levels of matching, such as an exact match on a postcode being weighted more heavily than a match on a common surname.

Preprocessing and Normalization Tools

The effectiveness of any name matching algorithm is heavily dependent on the quality of the input data. Python offers specialized libraries for domain-specific normalization:

- **cleancourt:** Designed for legal party names, this library standardizes company names

and distinguishes between individuals and organizations.
- **HumanMint:** Targeted at government and civil servant data, it normalizes job titles and department names through fuzzy matching and canonicalization.
- **pyjanitor:** Extends the Pandas API with a declarative syntax for cleaning column names and handling missing values, which is essential before performing record linkage.

The standard practice for name normalization involves lowercasing, removing non-alphanumeric characters, and collapsing multiple spaces into a single separator. The re (regular expression) module is frequently used to implement these patterns efficiently.

Machine Learning and Transformer-Based Entity Resolution

The most significant advancement in name matching in the 2024–2025 period is the shift from lexical similarity (matching characters) to semantic similarity (matching meaning) using transformer-based models.

Bi-Encoders vs. Cross-Encoders

The Sentence Transformers framework supports two primary architectures for name matching, each with distinct trade-offs in accuracy and scalability.

Bi-Encoders process each name independently to generate a fixed-dimensional vector embedding. These embeddings can be pre-computed and stored in a vector database. Similarity is calculated using cosine similarity or dot product between these vectors. This approach is highly scalable, enabling the comparison of a search query against millions of entries in milliseconds. Models like all-MiniLM-L12-v2 or EmbeddingGemma are frequently used in this capacity.

Cross-Encoders take a pair of names and process them simultaneously. This allows the model's attention mechanism to analyze the interactions between every token in both strings. While far more accurate at capturing nuance—such as distinguishing between "John Smith" and "Smith John"—cross-encoders do not produce embeddings and require a forward pass for every unique pair, making them computationally impractical for large datasets (O(n^2) complexity).

Aspect	Bi-Encoder (Retrieval)	Cross-Encoder (Reranking)
Input Format	Single sentence	Sentence pair
Output	Vector embedding	Similarity score (0–1)
Speed	High (Pre-computable)	Low (Pairwise processing)
Accuracy	Moderate (Semantic)	High (Nuanced interactions)
Use Case	Searching 1,000,000+ records	Reranking top 10–100 candidates

Specialized Embeddings and Benchmarks

In clinical and biomedical domains, specialized transformer models have demonstrated superior performance over traditional text matching. A 2024 benchmark using the World Health Organization (WHO) Classification of Tumours found that embedding-based methods achieved approximately 69% accuracy in terminology standardization, while traditional text-matching methods peaked at only 32.6%. Open-source models like jina-embeddings-v2 and snowflake-arctic have become popular for enterprise reconciliation tasks due to their support for

long context windows and multilingual data.

Architectural Best Practices: A Hybrid Implementation Strategy

Developing a best-in-class name matching system requires a tiered architecture that balances the efficiency of SQL with the precision of Python and Machine Learning.

The Pipeline Workflow

The most effective systems follow a standard workflow:

1. **Normalization:** Clean and standardize text in the database using SQL functions or specialized Python libraries.
2. **Blocking/Indexing:** Use SQL trigram indexes or Python blocking rules to reduce the search space from billions of pairs to thousands of high-probability candidates.
3. **Scoring:** Apply character-based (Jaro-Winkler) and semantic (Bi-Encoder) scores to the candidates.
4. **Thresholding and Review:** Categorize results into "Auto-match," "Manual Review," and "Non-match" based on confidence scores. Human-in-the-loop systems like Snowflake's harmonization app provide an audit trail for these decisions.

Example Implementation: PostgreSQL and RapidFuzz

A typical enterprise implementation might involve a SQL-based retrieval step followed by a Python-based scoring step for high precision.

PostgreSQL: Candidate Retrieval Script

The following SQL demonstrates the use of trigram indexes to efficiently retrieve potential name matches from a large table.

```
-- Step 1: Initialize the database with trigram support
CREATE EXTENSION IF NOT EXISTS pg_trgm;

-- Step 2: Create a table for target entities
CREATE TABLE customer_names (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    normalized_name TEXT GENERATED ALWAYS AS
(lower(regexp_replace(name, '[^a-zA-Z0-9 ]', '', 'g'))) STORED
);

-- Step 3: Create a GIN index for trigram similarity
CREATE INDEX idx_customer_names_trgm ON customer_names USING gin
(normalized_name gin_trgm_ops);

-- Step 4: Retrieve candidates for a search term
-- We use a similarity threshold to limit candidates
```

```
SELECT
    id,
    name,
    similarity(normalized_name, 'microsoft corp') as score
FROM customer_names
WHERE normalized_name % 'microsoft corp'
ORDER BY score DESC
LIMIT 50;
```

Python: High-Precision Scoring Script

Once candidates are retrieved, RapidFuzz can be used to perform more nuanced scoring in a Python application.

```
import pandas as pd
from rapidfuzz import fuzz, process, utils

# Sample candidates retrieved from the database
candidates =

search_term = "microsoft corp"

# Convert candidates to a list for RapidFuzz
choices = [c['name'] for c in candidates]

# Use Weighted Ratio (WRatio) for scoring
# WRatio combines Ratio, Partial Ratio, and Token Sort Ratio
results = process.extract(
    search_term,
    choices,
    scorer=fuzz.WRatio,
    processor=utils.default_process,
    limit=5
)

# Output results with scores
for match, score, index in results:
    original_id = candidates[index]['id']
    print(f"ID: {original_id} | Match: {match} | Score: {score:.2f}")

# Thresholding logic
for match, score, index in results:
    if score >= 90:
        print(f"Action: Auto-merge {match}")
    elif score >= 75:
        print(f"Action: Flag for manual review - {match}")
```

Example Implementation: Splink for Large-Scale Deduplication

For datasets where multiple columns (Name, DOB, Address) must be reconciled, Splink provides a more sophisticated probabilistic approach.

```
from splink.duckdb.linker import DuckDBLinker
import splink.comparison_library as cl
import pandas as pd

# Assume df is a pandas DataFrame with columns: first_name, surname,
dob, email
df = pd.read_csv("large_customer_dataset.csv")

# Define the Splink settings
settings = {
    "link_type": "dedupe_only",
    "blocking_rules_to_generate_predictions": [
        "l.first_name = r.first_name and l.surname = r.surname",
        "l.dob = r.dob"
    ],
    "comparisons":,
    "retain_intermediate_calculation_columns": True
}

# Initialize the DuckDB linker (highly efficient for single-node
processing)
linker = DuckDBLinker(df, settings)

# Estimate model parameters using Expectation Maximisation (EM)
# This learns the weights of the different fields without labeled data
linker.training.estimate_u_using_random_sampling(max_pairs=1e6)
linker.training.estimate_parameters_using_expectation_maximisation("l.
first_name = r.first_name")
linker.training.estimate_parameters_using_expectation_maximisation("l.
dob = r.dob")

# Run the prediction to find matches
df_predictions = linker.predict(threshold_match_probability=0.95)

# Cluster the results to assign a single ID to matched entities
df_clusters =
linker.clustering.cluster_pairwise_predictions_at_threshold(df_predict
ions, 0.95)

# Export results
df_clusters.as_pandas_dataframe().to_csv("deduplicated_customers.csv")
```

Insights on Governance and Operational Excellence

Entity resolution is not merely a technical problem; it is a governance challenge. The adoption of advanced name matching strategies must be accompanied by rigorous auditing and security practices.

Security and GDPR Compliance

Search operations, particularly in BigQuery and Snowflake, often trigger regulatory requirements for data lineage and audit trails. For General Data Protection Regulation (GDPR) reporting, organizations must be able to identify all rows associated with a specific individual. Advanced entity resolution ensures that these requests are comprehensive, catching variations that simple exact-match queries would miss. Furthermore, functions like AI_REDACT in Snowflake are increasingly used to strip personally identifiable information (PII) from text before it is processed by matching models.

Cost-Performance Trade-offs

The financial implications of different matching strategies are significant. While running a cross-encoder on every possible pair of names in a 100,000-row table would require computing nearly 5 billion comparisons—taking approximately 65 hours on high-end hardware—a bi-encoder can perform the same task in seconds. In cloud environments like Snowflake, the choice of warehouse size (e.g., 2X-LARGE for vector operations) must be balanced against the value of the insights generated.

Metric	SQL (Server-Side)	Python (Client-Side)	Hybrid (Recommended)
Data Movement	Zero (In-place)	High (Download/Upload)	Minimal (Filtered subsets)
Concurrency	High (Managed)	Low (Single process)	Moderate
Latency	Low	High (Network + CPU)	Optimized
Governance	Built-in RBAC/Audit	External control	Shared

Emerging Trends and Future Outlook

The field of name matching is moving toward a polyglot, AI-native stack. Several trends define the future of identity resolution:

1. **In-Database Machine Learning:** The integration of vector search and LLM functions directly into SQL engines (e.g., Snowflake Cortex, BigQuery Search) is reducing the need for external Python processing for standard tasks.
2. **Multilingual Phonetic Maturity:** Algorithms like Beider-Morse are becoming more accessible through libraries like Jellyfish, allowing for more equitable matching of global names.
3. **Matryoshka Representation Learning (MRL):** New embedding models, such as EmbeddingGemma, support truncated dimensions, allowing developers to balance storage costs and retrieval speed without significant loss in accuracy.
4. **Semantic Reranking:** The industry is standardizing on the bi-encoder/cross-encoder pipeline, using keyword and vector search for initial retrieval and specialized transformers

for final accuracy.

In summary, the best-in-class approach to name matching requires a nuanced understanding of the underlying data and the computational constraints of the environment. For massive datasets, SQL-based trigram and search indexing remains the foundation of efficiency. For complex, high-precision deduplication, the Python Splink framework offers unmatched statistical rigor. For the future of semantic identity, transformer-based embeddings and re-ranking pipelines provide the ultimate level of precision, bridging the gap between how machines read text and how humans understand identity.

Works cited

1. What is fuzzy search? Fuzzy search meaning. | Google Cloud, https://cloud.google.com/discover/what-is-fuzzy-search 2. Fuzzy Matching 101: The Complete Guide to Accurate Data Matching - Data Ladder, https://dataladder.com/fuzzy-matching-101/ 3. Exploring BigQuery Fuzzy Match Techniques - Cole Murray, https://murraycole.com/posts/bigquery-fuzzy-match 4. SQL vs. Python: A Comparative Analysis for Data - Airbyte, https://airbyte.com/data-engineering-resources/sql-vs-python-data-analysis 5. Data Preprocessing: SQL vs. Python - Data Column | Institute for Advanced Analytics, https://datacolumn.iaa.ncsu.edu/blog/2025/01/31/data-preprocessing-sql-vs-python/ 6. Top Programming Language Trends in Data Science: 2025 Insights - upGrad, https://www.upgrad.com/blog/programming-languages-trends-data-science/ 7. Fuzzy Name Matching Techniques | Babel Street, https://www.babelstreet.com/blog/fuzzy-name-matching-techniques 8. A Journey into BigQuery Fuzzy Matching — 2 of [1, ∞) — More Soundex and Levenshtein Distance | by Brian Suk | Google Cloud - Medium, https://medium.com/google-cloud/a-journey-into-bigquery-fuzzy-matching-2-of-1-more-soundex-and-levenshtein-distance-e64b25ea4ec7 9. String comparators - Splink, https://moj-analytical-services.github.io/splink/topic_guides/comparisons/comparators.html 10. Fuzzy String Matching in Python Tutorial - DataCamp, https://www.datacamp.com/tutorial/fuzzy-string-python 11. Comparison Library - Splink, https://moj-analytical-services.github.io/splink/api_docs/comparison_library.html 12. JAROWINKLER_SIMILARITY - Snowflake Documentation, https://docs.snowflake.com/en/sql-reference/functions/jarowinkler_similarity 13. Implementing Fuzzy Search in SQL Server Using New Inbuilt Functions - SQLServerCentral, https://www.sqlservercentral.com/articles/implementing-fuzzy-search-in-sql-server-using-new-in built-functions 14. Soundex & Fuzzy Logic: What They Are and When to Use Them, https://developer.bennysutton.com/blog/4089-soundex-fuzzy-logic-what-they-are-and-when-to-use-them 15. Documentation: 18: F.16. fuzzystrmatch — determine ... - PostgreSQL, https://www.postgresql.org/docs/current/fuzzystrmatch.html 16. An overview of DIFFERENCE and SOUNDEX SQL functions - SQLShack, https://www.sqlshack.com/an-overview-of-difference-and-soundex-sql-functions/ 17. SQL vs. Python: Frenemies of the Data World - The New Stack, https://thenewstack.io/sql-vs-python-frenemies-of-the-data-world/ 18. Documentation: 9.1: GiST and GIN Index Types - PostgreSQL, https://www.postgresql.org/docs/9.1/textsearch-indexes.html 19. Difference between GiST and GIN indexes - Stack Overflow, https://stackoverflow.com/questions/28975517/difference-between-gist-and-gin-indexes 20. How to optimize DB that is running pg_trgm similarity function? : r/PostgreSQL - Reddit,

https://www.reddit.com/r/PostgreSQL/comments/1lg4nxr/how_to_optimize_db_that_is_running_pg_trgm/ 21. PostgreSQL GIN index slower than GIST for pg_trgm? - Stack Overflow, https://stackoverflow.com/questions/43008382/postgresql-gin-index-slower-than-gist-for-pg-trgm 22. Increase fault tolerance for SQL Soundex for fuzzy string matching - Stack Overflow, https://stackoverflow.com/questions/25863806/increase-fault-tolerance-for-sql-soundex-for-fuzzy-string-matching 23. CLR vs. T-SQL: Performance considerations - About Sql Server, https://aboutsqlserver.com/2013/07/22/clr-vs-t-sql-performance-considerations/ 24. Fuzzy strings matching using Levenshtein algorithm on SQL Server (T-SQL vs CLR), https://pawlowski.cz/2010/12/28/sql_server-fuzzy-strings-matching-using-levenshtein-algorithm-t-sql-vs-clr/ 25. Optimizing a 'Fuzzy' Customer Search using Levenshtein Distance : r/dataengineering, https://www.reddit.com/r/dataengineering/comments/10vtu4k/optimizing_a_fuzzy_customer_search_using/ 26. Introduction to search in BigQuery - Google Cloud Documentation, https://docs.cloud.google.com/bigquery/docs/search-intro 27. Find approximate matches with fuzzy search | Spanner - Google Cloud Documentation, https://docs.cloud.google.com/spanner/docs/full-text-search/fuzzy-search 28. All functions (alphabetical) | Snowflake Documentation, https://docs.snowflake.com/en/sql-reference/functions-all 29. Snowflake Cortex AI Functions (including LLM functions), https://docs.snowflake.com/en/user-guide/snowflake-cortex/aisql 30. End-to-End Entity Resolution with Snowflake Cortex AI, https://www.snowflake.com/en/developers/guides/abt-bestbuy-entity-resolution/ 31. rapidfuzz/RapidFuzz: Rapid fuzzy string matching in Python using various string metrics - GitHub, https://github.com/rapidfuzz/RapidFuzz 32. RapidFuzz: Find Similar Strings Despite Typos and Variations - CodeCut, https://codecut.ai/rapidfuzz-rapid-string-matching-in-python/ 33. A Comparative Analysis of Python Text Matching Libraries: A Multilingual Evaluation of Capabilities, Performance and Resource Ut, https://ijeedu.com/index.php/ijeedu/article/download/188/99 34. (PDF) A Comparative Analysis of Python Text Matching Libraries: A Multilingual Evaluation of Capabilities, Performance and Resource Utilization - ResearchGate, https://www.researchgate.net/publication/390846511_A_Comparative_Analysis_of_Python_Text_Matching_Libraries_A_Multilingual_Evaluation_of_Capabilities_Performance_and_Resource_Utilization 35. Performance — Python Record Linkage Toolkit 0.15 documentation, https://recordlinkage.readthedocs.io/en/stable/performance.html 36. Deduplicating and linking large datasets using Splink - Real World Data Science, https://realworlddatascience.net/applied-insights/case-studies/posts/2023/11/22/splink.html 37. Defining Splink models - Splink, https://moj-analytical-services.github.io/splink/topic_guides/splink_fundamentals/settings.html 38. Super-fast deduplication of large datasets using Splink and DuckDB, https://www.robinlinacre.com/fast_deduplication/ 39. Deduplicating 7 million records in two minutes with Splink | by Robin Linacre | Data Science Collective | Medium, https://medium.com/data-science-collective/deduplicating-7-million-records-in-two-minutes-with-splink-4b1a87035a85 40. cleancourt - PyPI, https://pypi.org/project/cleancourt/ 41. I just published HumanMint, a python library to normalize & clean government data - Reddit, https://www.reddit.com/r/PythonProjects2/comments/1p8umyw/i_just_published_humanmint_a_python_library_to/ 42. Python Libraries for Data Clean-Up - StrataScratch, https://www.stratascratch.com/blog/python-libraries-for-data-clean-up/ 43. Speeding up name matching task : r/learnpython - Reddit, https://www.reddit.com/r/learnpython/comments/1f7npg9/speeding_up_name_matching_task/

44. Text Data Cleaning: Techniques for Preprocessing and Normalization, https://dataheadhunters.com/academy/text-data-cleaning-techniques-for-preprocessing-and-normalization/ 45. Names and normalization - Python Packaging User Guide, https://packaging.python.org/en/latest/specifications/name-normalization/ 46. Benchmarking Transformer Embedding Models for Biomedical Terminology Standardization, https://pubmed.ncbi.nlm.nih.gov/40718094/ 47. A Domain-Finetuned Semantic Matching Framework Based on Dynamic Masking and Contrastive Learning for Specialized Text Retrieval - MDPI, https://www.mdpi.com/2079-9292/14/24/4882 48. Bi-encoders and Cross-encoders for Sentence Pair Similarity Scoring (Part 1), https://www.dailydoseofds.com/bi-encoders-and-cross-encoders-for-sentence-pair-similarity-scoring-part-1/ 49. What is the difference between using a Sentence Transformer (bi-encoder) and a cross-encoder for sentence similarity tasks? - Milvus, https://milvus.io/ai-quick-reference/what-is-the-difference-between-using-a-sentence-transformer-biencoder-and-a-crossencoder-for-sentence-similarity-tasks 50. Sentence Embeddings. Cross-encoders and Re-ranking – hackerllama - GitHub Pages, https://osanseviero.github.io/hackerllama/blog/posts/sentence_embeddings2/ 51. A Deep Dive into Cross Encoders and How they work - Ranjan Kumar, https://ranjankumar.in/%F0%9F%94%8E-a-deep-dive-into-cross-encoders-and-how-they-work/ 52. Generate Embeddings with Sentence Transformers | Gemma - Google AI for Developers, https://ai.google.dev/gemma/docs/embeddinggemma/inference-embeddinggemma-with-sentence-transformers 53. Cross-Encoders — Sentence Transformers documentation, https://sbert.net/examples/cross_encoder/applications/README.html 54. Benchmark of 11 Best Open Source Embedding Models for RAG - Research AIMultiple, https://research.aimultiple.com/open-source-embedding-models/ 55. Cortex Search | Snowflake Documentation, https://docs.snowflake.com/en/user-guide/snowflake-cortex/cortex-search/cortex-search-overview