Frost: Benchmarking and Exploring Data Matching Results

Experiment, Analysis & Benchmark

Martin Graf
martin.graf@student.hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Lukas Laskowski
lukas.laskowski@student.hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Florian Papsdorf
florian.papsdorf@student.hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Florian Sold
florian.sold@student.hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Roland Gremmelspacher
roland.gremmelspacher@sap.com
SAP SE
Walldorf, Germany

Felix Naumann
felix.naumann@hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Fabian Panse
fabian.panse@uni-hamburg.de
Universität Hamburg, Germany

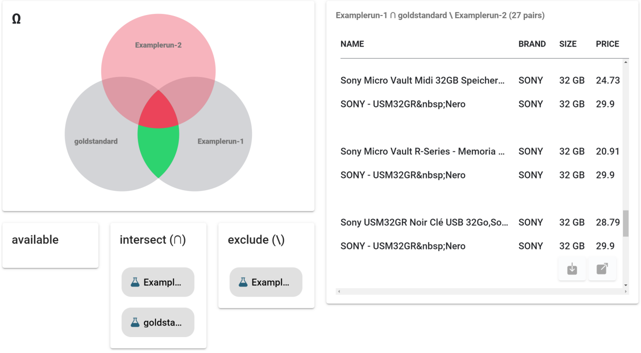
ABSTRACT

"Bad" data has a direct impact on 88% of companies, with the average company losing 12% of its revenue due to it [17]. Duplicates – multiple but different representations of the same real-world entities – are among the main reasons for poor data quality. Therefore, finding and configuring the right deduplication solution is essential. Various data matching benchmarks exist which address this issue. However, many of them focus on the quality of matching results and neglect other important factors, such as business requirements. Additionally, they often do not specify how to explore benchmark results, which helps understand matching solution behavior.

To address this gap between the mere counting of record pairs vs. a comprehensive means to evaluate data matching approaches, we present the benchmark platform Frost. Frost combines existing benchmarks, established quality metrics, a benchmark dimension for soft KPIs, and techniques to systematically explore and understand matching results. Thus, it can be used to compare multiple matching solutions regarding quality, usability, and economic aspects, but also to compare multiple runs of the same matching solution for understanding its behavior. Frost is implemented and published in the open-source application Snowman, which includes the visual exploration of matching results.

PVLDB Reference Format:
Martin Graf, Lukas Laskowski, Florian Papsdorf, Florian Sold, Roland Gremmelspacher, Felix Naumann, and Fabian Panse. Frost: Benchmarking and Exploring Data Matching Results. PVLDB, 15(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/HPI-Information-Systems/snowman.



Examplerun-1 ∩ goldstandard \ Examplerun-2 (27 pairs)

NAME	BRAND	SIZE	PRICE
Sony Micro Vault Midi 32GB Speicher...	SONY	32 GB	24.73
SONY - USM32GR Nero	SONY	32 GB	29.9
Sony Micro Vault R-Series - Memoria ...	SONY	32 GB	20.91
SONY - USM32GR Nero	SONY	32 GB	29.9
Sony USM32GR Noir Clé USB 32Go,So...	SONY	32 GB	28.79
SONY - USM32GR Nero	SONY	32 GB	29.9

Figure 1: Intersection of multiple experiments. This figure shows which ground truth duplicates Examplerun-1 found but Examplerun-2 did not.

1 DATA MATCHING

Most if not all businesses and organizations rely heavily on information systems and store their structured data in databases. These databases often contain errors, such as outdated values, typos, or missing information. According to Econsultancy "the average organization estimates that 22% of all its contact data is inaccurate in some way" leading to huge costs and non-monetary damage [17]. Efforts have been made in the past decades to better understand the different aspects of data quality [1, 51]. One prominent aspect of inaccurate data is (fuzzy) duplicates – the presence of multiple but different records representing the same real-world entity. Beyond poor data quality, duplicates exist in even more situations, for example during data integration pipelines. In this case, real-world entities can be interpreted in multiple ways. For example, a database containing details about persons could be matched by household, or on a per-person basis.

To address the issue of duplicates, various commercial and research systems to detect such duplicates have been developed [10, 14]. Systems detecting duplicates are generally referred to as (data) matching solutions, deduplication solutions, or entity resolution

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

systems. They can be categorized into different groups. The most popular are rule-based and machine learning solutions. Rule-based solutions are configured by hand-crafted matching rules to detect when a pair of records is a duplicate. An example rule in the context of a customer dataset could state that a high similarity of the surname is an indicator for duplicates, but a high similarity of customer IDs is not. Supervised machine learning models, on the other hand, are trained by domain experts who label example pairs from the dataset as *duplicate* or *non-duplicate*.

1.1 A Data Matching Benchmark Platform

To find the best matching solution for a specific use case and to help configure it optimally, different benchmarks for comparing matching results have been developed. Typical data matching benchmarks consist of a dataset, a ground truth annotation, and sometimes information on how to evaluate the performance. To answer the question of which matching solution is best, all competing matching solutions are run against the dataset. Then, the results are compared to the ground truth annotation. Finally, performance metrics scores, such as precision, recall, and f1 score are determined for each matching solution and contrasted.

On the one hand, many such benchmarks exist. On the other hand, they are scattered across different sources and sometimes do not allow easy comparability. Additionally, almost all existing matching solution evaluation techniques focus on the quality of matching solution results. However, aspects other than matching quality, such as business factors, also affect their usefulness in real-world deployment scenarios. In addition, traditional metrics often provide only a quantitative overview over the performance of matching solutions. Qualitative analyses, such as behavioral analytics for matching solutions, are scarce in the matching context – despite having a high relevance for common use cases, such as fine-tuning a matching solution. The straightforward but tedious way to gain qualitative insights consists of manually inspecting the result set of an experiment. However, since real-world datasets can contain millions of records, this becomes infeasible in practice.

To address these issues, we make the following contributions to the field of data matching benchmarks:

- **Exploration**: We propose techniques for systematically exploring and understanding matching results allowing qualitative inter-system and intra-system comparisons (see Section 4 and the example in Figure 1).
- **Soft KPIs**: We extend on effort measurements by aggregating relevant business factors, for example purchase costs and deployment type (see Section 3.3).
- **Benchmark platform**: We combine existing benchmarks, traditional quality metrics, a benchmark dimension for soft KPIs, and techniques to systematically explore and understand matching results into a benchmark platform called Frost. Frost can be used to compare multiple matching solutions regarding quality, usability, and economic aspects, but also to compare multiple runs of the same matching solution for understanding its behavior.
- **Snowman**: We implemented and published Frost in the open-source application Snowman[1] (see Section 5.1).

1.2 Formal Matching Process

In this section, we define the formal matching process and steps involved. Also, we introduce abbreviations and variables that will be used throughout the paper.

A dataset D is a collection of records which may contain duplicates. A record pair is a set of two records $\{r_1, r_2\} \subseteq D$. We denote the set of all record pairs in D as $[D]^2 = \{ A \subseteq D \mid |A| = 2 \}$. A matching solution M is a function which takes a dataset D as input and outputs a clustering $\{C_1, C_2, ...\}$ of D where all C_i are pairwise disjoint. We call the output of a matching solution *experiment*.

The pairs within each cluster C_i are predicted to be duplicates by the matching solution and called matches. All other pairs in $[D]^2$ are predicted to be no duplicates by the matching solution and called non-matches. Accordingly, a different representation for the clustering is a set of all matches $E \subseteq [D]^2$. E can be seen as a graph which has nodes for every record $r \in D$ and edges between all record pairs $\{r_1, r_2\} \in E$ (also called *identity link network* [34]). Because E represents a clustering of D, the graph is generally transitively closed. Thereby, we ensure that if r_1 and r_2 are matches and r_2 and r_3 are matches, r_1 and r_3 are considered to be matches too. Nevertheless, some real-world matching solutions output subsets of $[D]^2$ which are not transitively closed. Although their result set could easily be transitively closed, this step often introduces too many false positives in practice [20, 31]. Therefore, a clustering algorithm specific to the use case, such as described by Hassanzadeh et al. [31] or Draisbach et al. [20], needs to be applied to their result sets.

While it is most common to evaluate the performance of matching solutions only via their final results, typical matching solutions consist of multiple steps. Measuring the performance between these steps can provide useful insights for tweaking specific parts of the matching solution and helps to find bottlenecks of matching performance. We list the following steps for a typical data deduplication process [47]:

1. **Data preparation**: Segment, standardize, clean, and enrich the original dataset [42].
2. **Candidate generation**: Find a small subset of candidate pairs which contains as many real duplicates as possible [11, 48].
3. **Similarity-based attribute value matching**: Compute the similarities between the records' attribute values for each candidate pair [10, 19].
4. **Decision model / classification**: Given the similarities for each candidate pair, decide which candidate pairs are probably duplicates [10, 19]. In many cases, this step produces a final similarity or confidence score for each candidate pair. A pair is matched if its similarity score is higher than a specific threshold. From now on, we use the term *similarity* to refer to both similarity and confidence.
5. **Duplicate clustering**: Given the set of high probability duplicates, cluster the original dataset into sets of duplicates [20, 31].
6. **Duplicate merging / record fusion**: Merge the clusters of duplicates into single records [6, 18, 32]

Additionally, during data integration, matching solutions can be used to combine data from two or more sources. If the schemes of

[1] https://github.com/HPI-Information-Systems/snowman

the data sources differ, the matching solution may perform schema matching to increase the quality of its results [19].

In the following sections, we first situate our paper within related work (see Section 2). Afterwards, we discuss different means for benchmarking, including traditional quality measures and soft KPIs (see Section 3). Then, we introduce different evaluation techniques that allow for qualitative insights about matching solutions (see Section 4). Finally, we showcase Snowman, our implementation of Frost, and present a short study on the impact of effort measurements (see Section 5).

2 RELATED WORK

In this section, we outline existing work on benchmarking and exploring matching solutions. First, we discuss prominent benchmarks, then approaches that yield insights similar to our exploration techniques, and lastly other benchmark platforms.

2.1 Data Matching Benchmarks

Benchmarks can be used to evaluate the performance of matching solutions. Thus, they lay the foundation for major parts of our benchmark platform Frost. We discuss benchmarks in detail in Section 3.1. A list of prominent benchmarks, generators, and pollutors is collected in [47]. Below, we discuss two recent benchmarks that are especially relevant to Frost. Additionally, we regard methods to profile benchmarks.

In 2014, Saveta proposed SPIMBench, the *Semantic Publishing Instance Matching Benchmark* [55]. SPIMBench consists of a data generator capable of value, structural, and logical transformations producing a *weighted gold standard* and a set of metrics for evaluating matching performance. The weighted gold standard includes a history of which transformations have been applied to generated duplicates. Accordingly, it can be used to perform detailed error analysis.

Most recently Crescenzi et al. proposed a new and flexible schema matching and deduplication benchmark called Alaska [15]. The authors profiled the Alaska datasets with traditional profiling metrics and three new metrics for measuring heterogeneity, namely *Attribute Sparsity*, *Source Similarity*, and *Vocabulary Size*. In work from 2020 Primpeli and Bizer focused solely on profiling benchmark datasets and grouped 21 benchmarks according to five profiling metrics, namely *Schema Complexity*, *Textuality*, *Sparsity*, *Development Set Size*, and *Corner Cases*, into distinct categories [50]. Such profiling metrics allow for better comparability of quality metrics from different benchmarks because the profiled factors can be considered. Moreover, profiling metrics that measure difficulty or heterogeneity of datasets are a crucial step towards finding representative datasets for a given matching task when no ground truth annotations exist.

Frost supports a wide range of profiling metrics for measuring how similar a benchmark dataset and a real-world dataset are (see Section 3.2.4). Additionally, we use the notion of attribute sparsity proposed by Crescenzi et al. for classifying errors of matching solutions (see Section 4.4).

2.2 Exploration Opportunities

To our knowledge, there has been surprisingly little work on structured techniques for exploring and understanding matching results. Rostami et al. presented the tool SIMG-VIZ [52], which interactively visualizes large similarity graphs and entity resolution clusters. Thereby, it helps users to detect errors in the duplicate clustering stage. This is useful for improving the clustering algorithm and can give a comprehensive overview on the matching result. On the other hand, only a limited number of possible errors are highlighted and large graphs easily overwhelm users. To counteract these problems, we propose techniques that help users to detect errors within the decision model. Specifically, we reduce the amount of information presented to the user by filtering out irrelevant data, sorting it by interestingness, and enriching it with useful information about the type of error.

Elmagarmid et al. introduce additional investigation techniques for their rule-based approach as part of their tool NADEEF/ER [24]. With NADEEF/ER, they offer users a complete suite for rule-based entity matching including an exploration dashboard to analyze matching results, in example the influence of each individual rule on the result. Compared to NADEEF/ER, Frost uses a more generic approach to evaluate matching results, as it supports a broad variety of matching approaches.

A somehow related field to exploring matching results is training matching solutions by active learning. Matching solutions utilizing active learning, such as proposed by Sarawagi and Bhamidipaty [54], try to minimize review cost by asking human annotators only about uncertain matching decision. Because supervisors are shown only uncertain matching decisions, they can understand weaknesses of the matching solution.

2.3 Benchmark Platforms

One of the first data matching benchmark platforms has been proposed by Weis et al. [62]. The authors described a benchmark concept for XML data measuring effectiveness (matching quality) and efficiency (run-time). Frost integrates both effectiveness and efficiency measurements, but is not limited to them.

A new measurement dimension, *effort*, was proposed by Köpcke et al. in their benchmark platform FEVER [39]. Next to quality metrics, such as precision and recall, FEVER allows measuring the effort to configure a matching solution run by specifying labeling and parametrization effort. These KPIs can be compared to each other in effort-metric diagrams, answering questions such as “How much effort is needed to reach 80% precision?” Frost builds on this idea by integrating business requirements with the goal of supporting the decision-making process of selecting a matching solution. Concretely, it allows the comparison of matching solutions based on context-sensitive effort measurements, but also on further KPIs, such as deployment type and costs.

In later work, the authors used FEVER to evaluate different matching solutions. They found that “some challenging resolution tasks such as matching product entities from online shops are not sufficiently solved with conventional approaches based on the similarity of attribute values” [41]. This insight emphasizes the need for a comprehensive benchmark platform and the ability to systematically explore matching results.

Some data matching execution frameworks also work as (partial) benchmark platforms. *DuDe* is a modular duplicate detection toolkit [21], consisting of six components to facilitate the entire matching process. One of those components, the postprocessor, can evaluate the performance of the matching solution in each run. For this purpose, metrics, such as precision and recall, are calculated. This reduces the feedback loop between running a matching solution and interpreting performance results, and allows comparing different experiments performed with DuDe. On the other hand, DuDe does not support general comparability between matching solutions, because many matching solutions use other matching frameworks or do not use a framework at all.

3 BENCHMARKING DATA MATCHING SOLUTIONS

A duplicate detection benchmark typically consists of:

- One or more dirty *datasets* containing duplicates. These duplicates can be within (i.e., intra-source duplicates), but also between the individual datasets (i.e., inter-source duplicates).
- A *gold standard* modeling the ground truth (i.e., the correct duplicate relationships) between the given data records.
- A set of *quality metrics* that are used to evaluate the given matching solutions. These can be metrics that compare the matching solutions with the gold standard, such as recall or precision [45], but also metrics that measure some inherent properties of the solutions, e.g., the number of pairs that are missing to transitively close the set of discovered matches or some soft KPIs, such as the effort that is needed to compute it (see Section 3.3).

Moreover, such a benchmark can contain additional information that may help a matching solution to detect the duplicates, e.g., an occupancy history that models the address changes of persons [59] as is the case in the ERIQ benchmark [58].

3.1 Benchmark Datasets

A good benchmark should meet a number of conditions. Most importantly, its ground truth annotation should be as correct and complete as possible (see Section 3.1.1).

Second, to transfer evaluation results from this to other datasets, its data and error patterns must be real or at least realistic (see Section 3.1.2).

Third, the dataset should contain some so-called *corner cases* [50] to push matching solutions to their limits and reveal their quality differences. Furthermore, the dataset must be compatible with the objectives of the evaluation. For example, evaluating a matching solution focused on scalability requires a large dataset with millions of records, while the evaluation of a clustering algorithm requires a dataset with duplicate clusters beyond pairs of two records.

3.1.1 Gold Standards. To measure the correctness of an experiment, we need a reference solution with which we can compare the result of the experiment. This solution is also called *gold standard* or *ground truth* and should accurately reflect the true state of the real-world scenario as defined by the use case (e.g., matching by household vs. by person).

The truth about the correct duplicate relationships between the records of a dataset D can be captured in different ways. The most common approach is to store a list of all pairs of duplicate records (or their IDs respectively) in a separate file. The gold standard, however, typically represents complete knowledge about the correct duplicate relationships and thus corresponds to a final matching solution [45], i.e., it is a clustering of D where every record belongs to exactly one cluster. Thus, the gold standard can also be modeled within the actual dataset by adding an extra attribute that associates each record with its corresponding duplicate cluster.

In addition to the gold standard, there are other concepts for modeling a ground truth, such as a silver or an annealing standard, which aim to deal with incomplete and uncertain knowledge about the correct duplicate relationships [61].

3.1.2 Reference Datasets. Many users who need a benchmark platform have their own use cases with their own dataset. Since the true duplicate relationships within these datasets are usually unknown (this is, after all, the reason matching solutions are applied), the performance of a matching solution cannot be evaluated on the whole dataset of the use case itself. Instead, the evaluation is frequently performed on a small subset of the dataset or on a similar reference dataset (see Section 3.2.4).

Reference datasets can originate from the real world or can be artificially created. The advantage of real-world datasets is that they contain data collected from actual applications, with real errors and duplicates. However, as with the use case data, the disadvantage is that the true duplicate relationships within these records are usually not known in advance and must first be identified. There are three ways to annotate the record pairs of a real-world dataset with their duplicate status. The first one is to have the pairs manually reviewed by domain experts or a crowd. This process is expensive (in time and money), so it can only be applied to small datasets. Examples are the Cora [21] or the CDDB dataset [21]. The second approach is to integrate several (usually two) real-world datasets based on a common identifier (e.g., the social security number or the ISBN). An example of a dataset created in this way is the Abt-Buy dataset [40]. Those datasets, however, rarely have duplicate clusters with more than two records. The last approach is to annotate the duplicates semi- or fully automatically by applying an already existing matching solution. An example of a dataset annotated in this way is the NCVR dataset by Christen [12]. The obvious shortcoming of this approach is that the annotations are not guaranteed to be correct, and evaluation results will be biased towards matching solutions similar to those used for computing the annotations.

Alternatively, one can create an artificial dataset. There are two main means to do so: polluting existing datasets and synthesizing new datasets. The idea of data *pollution* [35] is to inject duplicates, errors, and different forms of presentation into one or more already existing clean dataset(s). On the one hand, users have full control over the number and distribution of inserted duplicates and errors, and can thus generate benchmark data suitable for their purposes. On the other hand, the generation of realistic errors and error patterns is by no means trivial, but its success plays a significant role in the suitability of the polluted data.

The input to a pollution process can – in theory – be any dataset. The benefit of using a real-world dataset is that the data values and data patterns are real. The disadvantage is that the dataset may already contain duplicates, which are then not annotated as such in the resulting gold standard. Furthermore, real data may not be allowed to be used for privacy reasons. Especially in that case, the *synthesis* of an artificial input dataset may be helpful [25]. The disadvantage here is the difficulty to achieve realistic data values and patterns. In particular, the similarity of non-duplicates has a large impact on the difficulty of the matching problem, because the detection of duplicates is much easier if a simply calculated similarity allows a clear separation of both classes and becomes the more challenging the more often similar records are non-duplicates.

Due to the possible customization of data pollution and data synthesis, it is logical to automate these processes. As described in [47], a variety of tools that perform this task already exist. Examples are GeCo [13], BART [2], or EMBench [36].

A great advantage of automatic generation tools is, that they can be used to generate a series of test datasets, which differ only in a single characteristic (e.g., the number of records, duplicates or errors) and are thus suitable for the systematic evaluation of the behavior of a matching solution with respect to the change of a specific condition. This helps answer questions like "How does the quality of the solution behave when the number of errors in the data increases?" To benefit from this capability, our benchmark platform Frost supports datasets generated by such tools and is able to evaluate matching solutions beyond individual test datasets.

3.2 Measuring Data Matching Quality

When ground truth annotations are available, a multitude of different metrics can be calculated. While some of them are generally used and considered essential, many more have been proposed and suit specific needs. To be universally useful but highly adaptable, Frost focuses on many well-known metrics but can be extended by any other metrics. We distinguish pair-based metrics and cluster-based metrics.

3.2.1 Pair-based Metrics. To compare an experimental result E against a ground truth annotation G of a dataset D as sets of pairs, the confusion matrix can be defined as shown in Figure 2.

This matrix allows the calculation of all metrics known from the context of binary classification (pair-based metrics). Pair-based metrics do not require the identity link network of experiment E to be transitively closed. Therefore, they can be used to calculate matching quality even at intermediate stages of the matching solution. For example, pair-based metrics allow to measure the performance of the candidate generation phase. Additionally, they facilitate directly contrasting the quality of matching solutions that return clusters with matching solutions that return pairs (and do not necessarily output transitively closed identity link networks) [62]. Note that pair-based metrics implicitly give more weight to larger clusters, as each pair of records within a cluster is counted towards the result.

	Positive	Negative
Predicted Positive	$E \cap G$ (TP)	$E \setminus G$ (FP)
Predicted Negative	$G \setminus E$ (FN)	$([D]^2 \setminus E) \setminus G$ (TN)

Figure 2: Confusion Matrix. Comparison of experiment result E against ground truth annotation G on dataset D as sets of pairs.

Another weakness of pair-based metrics is the fact that in the real-world there is almost always a large imbalance between true positives and true negatives (called class imbalance) [10]. While a dataset of n tuples usually contains only $O(n)$ duplicate pairs, it may consist of up to $O(n^2)$ non-duplicate pairs. Metrics that judge upon correctly classified non-duplicates (true negatives) are therefore considered unreliable. For example, the *accuracy* of matching results compared to a ground truth might be close to 1, even when all record pairs were classified as non-duplicates.

Frost supports a wide selection of pair-based metrics considering the above observations (there are many other pair-based metrics and Frost can easily be extended with them):

- **Precision**: How many matches are true duplicates?
- **Recall**: How many true duplicates are matches?
- **f1 score**: f1 score is the harmonic mean between precision and recall, for which Hand and Christen note the following flaw: the relative importance assigned to precision and recall depends on the linkage method being used and not the actual problem as it should be [29].
- **f* score [30] / critical success index [56]**: Suggested as a better alternative to f1 score in recent work, f* score (also known as critical success index or threat score) is the proportion of true duplicates combined with matches which are correct (true duplicate and match).
- **False negative rate**: How many true duplicates are non-matches?
- **False discovery rate [4]**: How many matches are true non-duplicates?
- **Fowlkes-Mallows index [27]**: Folwkes-Mallows index is the geometric mean between precision and recall and was initially developed as a similarity metric for clusterings.
- **Matthews correlation coefficient [9]**: A balanced measure of quality taking into account every sector of the confusion matrix. Although the set *true negatives* is used, Boughorbel et al. showed that the Matthews correlation coefficient can still be used and produces meaningful values [7].
- **Pairs completeness [38]**: How many true duplicates are selected as candidate pairs in the candidate generation phase?
- **Reduction ratio [38]**: Which fraction of record pairs has been eliminated in the candidate generation phase?

3.2.2 Cluster-based Metrics. Cluster-based metrics are most often computed using similarities between clusters of the ground truth and the experiment [3, 45, 46]. An advantage of cluster-based metrics is that they are immune to the class imbalance described above. On the other hand, they cannot be used to directly evaluate matching solutions that produce non-transitively closed sets of matches [62]. For example, the output of intermediate stages of a matching solution pipeline is usually not clustered.

Frost utilizes the following prominent cluster-based metrics. Again, there are many more prominent cluster-based metrics and Frost can easily be extended with them.

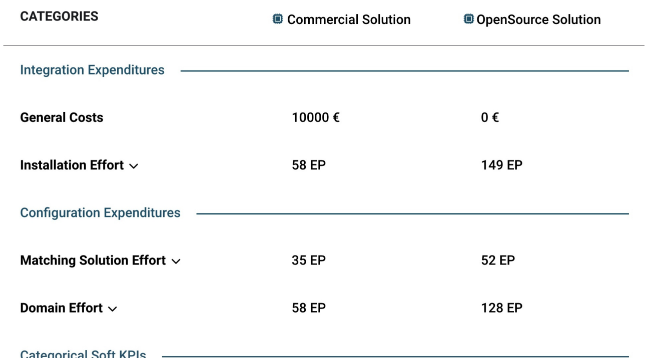
- **cc-Recall [5]**: Average maximal Jaccard similarity of every ground truth cluster to any experiment cluster.
- **cc-Precision [5]**: Average maximal Jaccard similarity of every experiment cluster to any ground truth cluster.

- **cc-f1 score [5]**: Harmonic mean between cc-Precision and cc-Recall.
- **Variation of information [44]**: The amount of information lost or gained from converting the experiment clustering into the ground truth clustering.
- **Generalized merge distance [45]**: Total costs of all cluster splits and merges necessary to transform one clustering into the other. Interestingly, it can be customized to calculate pairwise recall and precision as well as the variation of information.

3.2.3 Evaluating Quality Without Ground Truth. A few metrics try to estimate matching quality on real-world datasets without ground truth annotations. For example, the main idea of Idrissou et al. is that redundancy in identity link networks correlates with high matching quality [34]. Interestingly, their experiments show a "very strong predictive power of [. . . their] e_Q metric for the quality of [. . . identity link networks] when compared to human judgement" [34]. On the other hand, these metrics cannot substitute a ground truth, because they are unable to validate which matching decisions are correct. Thus, we primarily consider them for another application of Frost: finding a representative benchmark dataset for a real-world matching task for which no gold standard is available.

3.2.4 Finding a Representative Benchmark Dataset. As ground truth annotations are rarely available for real-world scenarios, it is useful to find benchmark datasets similar to a given use case dataset to estimate the performance of matching solutions on this dataset. Because no similarity measure for entity matching benchmarks and use case datasets currently exists, identifying similar benchmark datasets is difficult. To ease this process, Frost includes a list of factors impacting matching difficulty. It remains to the experts to determine how important these aspects are for their use cases, and to use them for evaluating how suitable a benchmark dataset emulates their use case dataset.

- **Domain**: The domain of use case dataset and benchmark dataset should match or be closely related.
- **Attribute type**: Benchmark dataset and use case dataset should have similar attribute types, for example mainly numerical or mainly categorical.
- **Tuple count**: In [22], Draisbach and Naumann showed that dataset size has influence on the optimal similarity threshold. Thus, using a benchmark dataset with similar size compared to the use case dataset may yield more representative results.
- **Number and size of duplicate clusters**: The amount and size of duplicate clusters in the ground truth annotation of the benchmark dataset should closely resemble the amount and size of duplicate clusters in the ground truth annotation of the use case dataset. Because the ground truth annotation for the use case dataset is unnokwn, the metrics have to be estimated. Heise et al. developed a method for this estimation [33].
- **Matching solution**: The matching solution itself may provide valuable insights into how similar benchmark dataset and use case dataset are. Relevant factors could include metrics for approximating quality without requiring a ground truth annotation from Section 3.2.3, the similarity of the clusterings of the matching solution on use case and benchmark dataset, and the number of pairs after classification which miss from the transitive closure. Note that some of these metrics require normalization if certain properties of the datasets, such as record count, do not match.



CATEGORIES	■ Commercial Solution	■ OpenSource Solution
Integration Expenditures		
General Costs	10000 €	0 €
Installation Effort ⌄	58 EP	149 EP
Configuration Expenditures		
Matching Solution Effort ⌄	35 EP	52 EP
Domain Effort ⌄	58 EP	128 EP
Categorical Soft KPIs		

Figure 3: Decision Matrix. An open-source and a commercial matching solution are compared in Frost's decision matrix. We aggregate effort into effort points to measure allocation of human resources, but it can also be viewed as "expertise" and "HR-amount".

- **Profiling metrics**: As discussed in Section 2.1, a few metrics have been proposed for profiling benchmark datasets. These may be worth considering.

We take a first step towards a similarity measure for entity matching benchmarks and use case datasets in Appendix C by measuring the impact of several of the above factors on matching performance.

3.3 Soft KPIs

Every matching solution has different advantages and disadvantages and requires a different type of configuration. As an example, supervised machine learning-based approaches need training data, whereas rule-based approaches need a set of rules. When deciding which matching solution to use for a specific use case, these properties are of importance, because they influence how expensive and time-consuming it is to employ the matching solution. To assist the decision process, Frost includes a benchmark dimension for soft KPIs, which models business aspects.

Below, we first explain how Frost measures effort. Then we discuss different soft KPIs included in Frost. Finally, we explain how Frost helps users to evaluate these KPIs (see Figure 3 for an example).

3.3.1 Measuring Effort. We use the term *effort* to measure the amount and complexity of work necessary to perform a specific task. In Frost, effort is estimated with the following variables:

- **HR-amount**: HR-amount represents the time an expert would need to finish the task in person-hours.
- **Expertise**: Expertise represents the skill of the expert regarding the task on a scale from zero (untrained) to 100 (highly skilled).

HR-amount and expertise are interdependent: When comparing two persons with different expertise, in average, the person with more expertise will perform faster. When effort has to be measured over time, for example when maintaining a matching solution over its lifetime, these variables can be measured on a monthly basis.

Frost can aggregate HR-amount and expertise into a scalar to allow easy comparability between matching solutions. Depending on the goal, there are different ways to achieve this:

- **Measuring monetary costs**: Expertise is typically related to pay level. Therefore, mapping expertise to pay level and then multiplying it with HR-amount yields a rough estimation of the cost of performing the task.
- **Measuring allocation of human resources**: In some cases, only a specific set of human resources may be available for performing the task. Accordingly, the goal is to minimize the allocation time of the available human resources. A rough estimation of this allocation time can be calculated by dividing HR-amount by the number of available people and adding the time it takes to train them until they have the required expertise.

In any case, the aggregation is very imprecise. Because of this, we prefer to work with effort as a two-dimensional vector of both HR-amount and expertise.

3.3.2 Lifecycle Expenditures. One important business aspect is the expenditure for integrating and operating a matching solution over its entire life-cycle. Based on life-cycle cost analysis (LCCA) [23], Frost defines the following soft KPIs to represent the different product phases:

- **General costs**: General costs are split into onetime and recurring and include all direct monetary costs over the entire life-cycle of the product. Frost views all costs under this single category because project specific costs vary widely depending on different factors, such as the payment model of the matching solution (e.g., one-time costs, monthly costs, pay-per-use). Note that this also includes the costs for disposing of the matching solution which are typically listed separately [23].
- **Integration effort**: Integration effort measures the effort it takes to get the matching solution ready for production within a companies ecosystem. This for example includes setting up the hardware, connecting the solution with the datasource, and testing the system.
- **Domain effort**: After integrating the matching solution into the companies software-ecosystem, it needs to be configured for its use case. Domain effort measures to which extent domain experts are needed for configuring the matching solution. For example, supervised machine learning-based approaches require training data which is created by domain experts classifying pairs of records as *duplicate* or *non-duplicate*.
- **Matching solution effort**: Matching Solution effort measures to which extent matching solution experts are needed for configuring the matching solution. For example, rule-based approaches often require matching solution experts to implement rule sets based upon the knowledge of a domain expert.

3.3.3 Categorical Soft KPIs. Next to lifecycle expenditures, there are a few more, mainly categorical aspects relevant for businesses. Below, we compiled a selection of such factors that are included in Frost.

- **Deployment type**: The deployment type describes in which ways the matching solution can be deployed, for example on-premise or cloud-based.
- **Interfaces**: This describes the user interfaces (e.g., GUI, API, CLI) supported by the matching solution.
- **Matching solution type**: Matching solution type categorizes the matching solution by its overall approach, such as rule-based, supervised machine learning, clustering, and probabilistic. Depending on the type, specific costs may arise (e.g., for labelling a training dataset).

3.3.4 Soft KPIs of Experiments. Next to measuring and evaluating soft KPIs on a matching solution level, Frost supports measuring and evaluating soft KPIs on an experiment basis:

- **Run-time**: Run-time measures how long the matching solution took to finish the deduplication process for this experiment. Next to plain time, this also includes the performance stats of the executing system.
- **Configuration effort**: This measures the effort for configuring the matching solution for this experiment.

3.3.5 Limits of Soft KPIs. The main goal of soft KPIs is to provide users a comparable overview on relevant, non-performance properties of matching solutions and experiments. Although the above discussed KPIs measure such aspects, some of them are estimated and therefore highly unreliable. Additionally, not every economic impact can be measured because some implications are unforeseeable. For example, our business partners remarked that often an estimation of the projected balance by using the cost of one false positive versus the cost of one false negative is not a good indicator.

While many non-effort KPIs are objective and therefore easily comparable, effort is subjective and has to be estimated. People with varying skill sets often have different opinions on how long it takes to configure a matching solution. We try to mitigate this problem in Frost by employing multiple techniques:

- **Measuring expertise**: As described above, Frost measures effort not only with HR-amount, but also with expertise. We do this to counteract the fact that people with different skill sets often estimate the lengths of tasks differently. Chatzoglou and Macaulay state that low experience is an indicator for increased time and cost and that experience is considered an important factor for productivity [8]. On the other hand, comparability is still not perfect: In 1993 Lakhanpal showed that to achieve high productivity in groups prior experience is not as important as for example group cohesiveness [43].
- **Measuring HR-amount in categories**: Frost supports measuring time in categories, such as minutes, hours, days, or weeks. This may be beneficial when multiple people participate in a benchmark workflow, because it compensates the

estimation mismatch between different persons. Nevertheless, this approach reduces the overall accuracy of individual time measurements.

3.3.6 Evaluating Soft KPIs. An evaluation of soft KPIs should provide users with a comparable overview on all relevant, non-performance properties of matching solutions and experiments. While aggregating soft KPIs into a single, representative non-performance metric might sound useful at first, this is not adequate to realistic use cases. Many of the discussed soft KPIs are categorical, and even the numeric KPIs have their own, sometimes hidden implications. As an example, even effort cannot be easily aggregated, because the required experts are different when the type of expertise is different.

That is why Frost supports two different evaluation techniques for matching solution soft KPIs. On the one hand, it utilizes a decision matrix including all above metrics side by side (such as in Figure 3). Importantly, this decision matrix also includes quality metrics to provide a holistic view on the attractiveness of the compared matching solutions. On the other hand, Frost provides users the possibility to aggregate metrics. For example, to estimate costs, the effort-based metrics could be converted into costs as described above and added to general costs. Because this aggregation depends on the use case, Frost does not define aggregation strategies on its own, but rather provides a framework for aggregating soft KPIs and quality metrics into use case specific KPIs.

As proposed and used by Köpcke et al. [39, 41], Frost aids users in analyzing soft KPIs for experiments with a diagram-based approach. This helps them to answer questions, such as how much effort they need to reach a specific metric threshold (e.g., 80% precision), whether increased run-time yields better results, or how good a matching solution is out of the box versus how much effort it takes to improve the results. The diagram gets especially interesting when experiments from multiple matching solutions are compared. Evaluations thereby become competitive and allow discovering different characteristics of the matching solutions.

4 EXPLORING DATA MATCHING RESULTS

The general workflow for improving matching solutions and arriving at a sufficient configuration is usually iterative. Thus, after one run has finished, its results need to be analyzed to gain insights about the solution's behavior. Afterwards, the matching solution can be refined accordingly and re-run. As motivated in Section 1.2, we present structured approaches to explore data matching results. Specifically, we reduce the amount of information presented to the user by filtering out irrelevant data, sorting it by interestingness, and enriching it with useful information about the type of error. Finally, we introduce diagram-based evaluations.

4.1 Set-based Comparisons

Manual inspection of experimental results can be an inconvenient experience – especially if the matching solution lacks a human-readable output format. As an example, some output formats consist solely of identifiers and thus require to be joined with the dataset to be helpful. Additionally, only limited information can be extracted by looking at results side-by-side as in practice usually more than two result sets are compared. A common use-case is to contrast multiple runs of the same matching solution with each other, or to evaluate differences between two distinct solutions and a ground truth.

Frost supports a generic set-based approach to result evaluation that enriches identifiers with the actual dataset record. The set operations *intersection* and *difference* can describe all partitions of the confusion matrix, as introduced in Section 3.2.1. As an exemplary evaluation, consider two result sets $E_1, E_2 \subseteq [D]^2$, where E_2 serves as ground truth. The subset of false positives is defined as the set of elements in E_1 that are not part of the ground truth E_2, or simply $E_1 \setminus E_2$. While the confusion matrix is limited to evaluating binary classification tasks with two result sets, the generic approach can compare multiple result sets.

As an intuitive visualization technique, Frost makes use of Venn diagrams. When n experiments are compared, these diagrams describe all $\binom{n}{2}$ possible subsets visually. In contrast to the generic set-based approach presented above, Venn diagrams are limited in the number of sets that can be feasibly visualized. As proven by Ruskey et al., Venn diagrams of more than three sets need to use geometric shapes more advanced than circles [53].

Set-based comparisons and Venn diagrams can help to answer a variety of evaluation goals.

- Compare two matching solutions' result sets against a ground truth to discover common pairs. This evaluation can easily be visualized with circle-based Venn diagrams.
- Find shortcomings or improvements of a new matching solution compared to a list of proven solutions by selecting all duplicate pairs only the new solution detected.
- Create an experimental ground truth [61] from the intersection of multiple experiments.

Because exploration is supposed to be interactive, an implementation should provide vivid Venn diagrams. Clicking on regions should allow to select the corresponding set intersection. Thereby, the desired configuration can be composed easily according to its visual representation.

4.2 Pair Selection Strategies

While set-based comparisons are useful on their own, real-world datasets can contain millions of records, making it unfeasible to further investigate as a whole. Therefore, strategies to reduce the number of pairs shown are crucial. Frost supports a wide range of selection techniques to highlight relevant pairs which can be used separately or as a composition according to the current use case.

4.2.1 Pairs around the Threshold. An easy section of the result to further investigate is located close to the similarity threshold, as it includes information on border cases. Pairs in this section are usually considered uncertain, as a slight shift of the threshold may change their state. Nevertheless, they still yield helpful insights about what is especially difficult for the matching solution. To select k pairs, one can either choose $\frac{k}{2}$ pairs above and below the threshold or based on a certain proportion. For instance, one interesting proportion is the ratio of incorrectly classified pairs above the threshold to below the threshold.

4.2.2 Incorrectly Labeled Outliers. Another group of interesting pairs lies further away from the threshold. For example, one could evaluate why the matching solution failed by searching for a common "misleading" feature among the selected pairs. Therefore, we allow selecting incorrectly labeled pairs that are the furthest away from the threshold.

4.2.3 Percentiles with Representatives. Sometimes the goal is to get an overview over the matching quality before diving into details. For this, we support finding representative pairs from all parts of the result set. Conceptually, this strategy sorts result sets by a similarity score and then splits them into smaller partitions. Each of these partitions is then reduced to a few representative pairs that represent the matching solution's behavior within this partition.

Let E be a result (sub)set with m pairs that is split into k equally-sized partitions. To sample b representative pairs for each partition, different choices exist:

- **Random sampling**: The selection of b pairs is sampled randomly from each partition. While this technique is unbiased, it may also only yield uninteresting pairs and thereby no helpful insights.
- **Class-based sampling**: For a partition with k_T correctly and k_F incorrectly classified pairs, we randomly sample $b * \frac{k_T}{k_T+k_F}$ correctly and $b * \frac{k_F}{k_T+k_F}$ incorrectly labeled pairs. Thereby, we make sure to weigh the numbers of pairs according to the algorithms performance.
- **Quantile sampling**: Alternatively, b pairs can be sampled by selecting b quantiles, again based on the similarity score. For $b = 5$, this would mean to select quantiles 0, 0.25, 0.5, 0.75, and 1. Compared to sampling around the median, this technique is unbiased and will represent different parts of the partition accordingly.

Additionally, we can label each partition with its confusion matrix and metrics. Thus, users can focus on those partitions with high error levels. A partition with few to no incorrectly labeled pairs is considered to be a confident section. In contrast, a section with many false positives and/or false negatives is very unconfident, and therefore deserves more attention.

4.2.4 Plain Result Pairs. As outlined in Section 1.2, Frost requires result sets to be transitively closed. On the one hand, this can lead to more realistic metrics. But on the other hand, it can also enlarge small result sets to a very large number of pairs and thereby possibly introduces a substantial number of false positives. Thus, Frost includes a selection strategy that will hide all pairs that were added by a clustering algorithm from a given result subset. What remains are all pairs that were originally labeled by a matching solution. To enable this, Frost requires information on which pairs were added during the clustering process and which were labelled by the matching solution itself.

4.3 Sorting Strategies

Besides reducing the result sets to smaller subsets, Frost also supports to sort pairs by their *interestingness* within a given subset. When relevant pairs are shown first, developers can gain insights more quickly to improve the matching solution's performance on a given dataset. The usefulness of the sorting procedure varies between strategies and use case. Below, we discuss several measures of interestingness of record pairs.

4.3.1 Similarity Score. A common score to rank any set of pairs is the similarity of a pair's records. Whenever similarity values are available for all pairs, this technique offers a view on the data from the matching solution's perspective.

4.3.2 Column Entropy. Additionally, we define independent scores that were not part of a matching solution's output. Sorting with such a score may yield additional insights.

For each token t within a given cell, let $prob_t$ be its occurrence probability within the cell and $columnProb_t$ the probability within the column. The cell entropy is calculated by:

$$\sum_{token\ t} prob_t \cdot -log(columnProb_t)$$

where the second factor describes a token's information content within its column. This formula is relatively close to the original definition of entropy by Shannon from 1948, but is applied column-wise [57]. For a given pair $p = \{r_1, r_2\}$, we can calculate its entropy as the sum of all cell entropies of both records. Pairs with a particularly high entropy score contain many rare tokens and are therefore expected to be easier to correctly classify. Depending on dataset and matching solution, we may observe a divergence in the distribution of entropy among the confusion matrix. If not, we can still use entropy as a score to sort pairs within a subset of the result set(s).

4.4 Error Analysis

To better understand why a pair was misclassified by a certain matching solution, one could analyze why a similar pair was labelled correctly. Thereby, one can gain insights on why the matching solution came to a false conclusion and find errors within the decision model. Frost allows to enrich a misclassified pair $p_f = \{e_{f,1}, e_{f,2}\}$ with a correctly classified pair $p_t = \{e_{t,1}, e_{t,2}\}$. We search for p_t by considering only correctly classified pairs and selecting the one which is most similar to p_f. We describe the similarity between the pairs p_f and p_t with vectors

$$\mathbf{v}_{direct} = \begin{pmatrix} sim(e_{f,1}, e_{t,1}) \\ sim(e_{f,2}, e_{t,2}) \end{pmatrix} \text{ and } \mathbf{v}_{cross} = \begin{pmatrix} sim(e_{f,1}, e_{t,2}) \\ sim(e_{f,2}, e_{t,1}) \end{pmatrix}$$

To compare these vectors with each other, we convert each one into a scalar distance measure. For this, the Minkowski metric with $q \in [1, 2]$ is used against $\vec{0}$ as the reference point:

$$distance(\mathbf{v}) = D(\mathbf{v}, \vec{0}) = (|\mathbf{v}_1 - 0|^q + |\mathbf{v}_2 - 0|^q)^{\frac{1}{q}}$$

For $q = 1$ this equals the Manhattan distance and for $q = 2$ the Euclidean distance. It depends on the user to choose $q \in [1, 2]$ depending on the use-case. Finally, we define the distance score of p_t against p_f as

$$score = \max\{distance(\mathbf{v}_{direct}),\ distance(\mathbf{v}_{cross})\}$$

Whichever candidate pair p_t scores highest is then selected.

To receive best results, all possible pairs should include a similarity score. Since this would require the matching solution to compare $O(n^4)$ values for a dataset of size n, a possible extension to Frost could be to calculate a simple distance measure for a set of promising pairs internally.

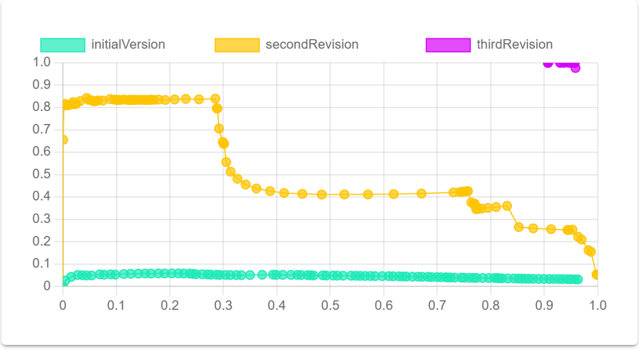


Figure 4: Precision-Recall Curve. This diagram plots recall against precision for a given set of similarity thresholds.

4.5 Diagram-based Exploration

All strategies so far are for set-based comparisons and either limited the amount of pairs shown (Sec. 4.2), sorted it (Sec. 4.3) or added additional information (Sec. 4.4). Here, we introduce a set of diagrams that aid in understanding a matching solution's behavior.

4.5.1 Metric/Metric Diagrams. For matching solutions that return similarity scores, one objective is to find a good similarity threshold. Frost utilizes metric/metric diagrams for this objective. Those diagrams compare two quality metrics against each other for a given set of similarity thresholds. A commonly known diagram is the precision/recall curve (see Figure 4). With it, one can visually observe which point (and thereby similarity score) yields the best ratio between both metrics. Another well-known diagram is the ROC curve [28] plotting sensitivity (*also:* recall) and specificity against each other. This may not be suitable for every use case though, as specificity depends on the count of true negatives. However, depending on the shape of the curve, these diagrams may yield insights upon both a good similarity threshold and the reliability of the matching solution. Next to using metric/metric diagrams in isolation, multiple diagrams between multiple matching solutions can be compared for competitive insights.

A limitation of this technique is that it heavily depends upon how many pairs have a similarity score assigned. Specifically, values further away from the matching solution's threshold may not be representative, as pairs with similarity scores significantly lower than the threshold may not be included in the result set.

The naïve approach to calculate metric/metric diagrams is to sample metrics at different similarity thresholds without reusing insights from other thresholds. This gets infeasible for large datasets which contain more than a few thousand records. To address this problem, we developed an efficient algorithm to compute metric/metric diagrams. We explain the algorithm in Appendix D.

4.5.2 Attribute Sparsity. As missing attribute values are known to influence and complicate matching tasks [15, 49, 50], we want to further investigate which attributes precisely affected a matching solution's performance most. Attribute Sparsity as introduced by Crescenzi et al. measures how often attributes are, in fact, populated within a given dataset [15]. Thereby, the authors profile a given dataset's difficulty together with additional profiling dimensions. Since we aim to profile a matching solution's result set instead, we define a metric that measures the influence of null-valued attributes by the count of incorrectly assigned labels as follows: Let D be a dataset and a be an attribute of D. We define $nullCount(a)$ as the count of pairs in $[D]^2$ where at least one record of the pair is null in attribute a. Additionally, we define $falseNullCount(a)$ as the count of incorrectly classified pairs in $nullCount(a)$ and $nullRatio(a)$ as:

$$nullRatio(a) = \frac{falseNullCount(a)}{nullCount(a)}$$

In contrast to the raw $nullCount$, immoderate amounts of null occurrences within an attribute do not bias the $nullRatio$. Calculating the metric for all attributes a in D yields a statistical distribution. Graphical representations that show scores for discrete buckets, such as bar charts, support comparing measured scores. Thereby we can observe the following: Attributes with higher $nullRatio$ scores are statistically more relevant for the matching decision as their absence could be related to more incorrectly assigned labels [50]. For instance, we observed that the ratio reveals a high significance for the attributes *author* and *title* in the Cora dataset [21] for the Magellan matching solution [37].

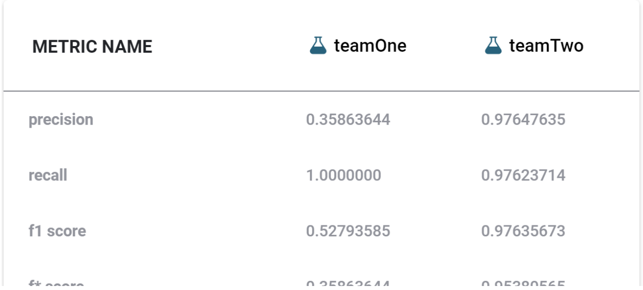
If the revealed significant attributes do not match the expectation, this likely comes down to one of two reasons:

- **Semantic mismatch**: A semantic mismatch exists if the matching solution weighs attributes heavily that are semantically irrelevant for a matching decision. For instance, a matching solution learned to weigh attributes b and c more significantly while a and b are more important in reality. A semantic mismatch is an indication that the provided rule set or the learned network's weights are not consistent with the domain of the given dataset.
- **Material mismatch**: A material mismatch exists if the statistically assumed significance of attributes is not adequate for the underlying dataset. For instance, a matching solution weighs attributes a and b while the underlying dataset is often null in these attributes. This mismatch might occur when a matching solution is used on another dataset than it was initially optimized for (for example due to cross-learning).

A downside is that $nullRatio$ relies on interspersed null values within the dataset D and a meaningful and sophisticated schema. Such a schema contains several attributes that provide meaningful information, for example street and city split instead of combined in a single address field. For instance, the Cora dataset fulfills the requirements with an average attribute sparsity of 0.58 and a schema with 17 attributes [21].

In conclusion, the exploration of $nullRatio$ allows insights into the matching solution's handling of null values.

4.5.3 Attribute Equality. Similar to attribute sparsity, Frost allows to investigate the influence of equal attribute values on the matching process, too. Equal attribute values can indicate a duplicate pair, although equality in one attribute is usually not a sufficient criterion. For instance, while attributes, such as the person's name, may be sufficient for a match, others, such as post code, may not. Therefore, Frost includes attribute equality as a dimension to statistically analyze which equal attributes are related to incorrectly assigned labels significantly often.



METRIC NAME	teamOne	teamTwo
precision	0.35863644	0.97647635
recall	1.0000000	0.97623714
f1 score	0.52793585	0.97635673
f* score	0.35863644	0.95389565

Figure 5: N-Metrics Evaluation. This page allows to compare quality metrics for multiple experiments at once.

Let D be a dataset and a be an attribute of D. First, we define $equalCount(a)$ as the count of pairs in $[D]^2$ where both records of the pair are equal in attribute a.

Additionally, we define $falseEqualCount(a)$ as the count of incorrectly classified pairs in $equalCount(a)$. We set:

$$equalRatio(a) = \frac{falseEqualCount(a)}{equalCount(a)}$$

A high $equalRatio(a)$ for a given attribute a indicates that the matching solution did not weigh the matching sufficiency of a correctly (either too high or too low).

Again, calculating the metric for all attributes a in D yields a statistical distribution which, if compared across all attributes, can yield helpful insights. Similarly, bar charts can be used as an evaluation tool.

5 DEPLOYMENT AND TOOLING

In this section, we present our implementation of Frost called Snowman and perform two example evaluations with it.

5.1 Snowman

We have implemented Frost in the application Snowman[2]. Next to traditional metric evaluation pages, Snowman has full support for our soft KPIs dimensions from Section 3.3 and supports the main exploration concepts from Section 4. Further, Snowman ships with a range of preinstalled benchmark datasets (including ground truths). This gives users the ability to easily evaluate and compare matching solutions in multiple domains out of the box.

To also allow automated usage, Snowman defines a REST API and command line interface through which all functionality can be accessed. The tool runs on all major operating systems and can be used by multiple users at once. For further details, see Appendix A about Snowman's back-end and Appendix B about its front-end.

Below, we present a selection of evaluations that are already part of Snowman. A full list can be found in Snowman's online documentation[3].

- **Data matching expenditures**: Snowman implements both the decision matrix depicted in Figure 3 and the diagram for evaluating experiment level expenditures as described in Section 3.3.6.
- **Set-based comparisons**: Snowman supports intersecting and subtracting experiments and ground truths with the help of an interactive Venn-diagram as described in Section 4.1 (see Figure 1). To enhance the evaluation process, Snowman shows complete records instead of only entity IDs. If only intersections operators are used, Snowman groups clusters.
- **Evaluating similarity scores**: Snowman helps users to find the best similarity threshold by plotting the metric/metric diagrams discussed in Section 4.5.1 (see Figure 4). It also allows to compare similarity functions of multiple matching solutions and multiple similarity functions of one matching solution.

5.2 SIGMOD Contest

The ACM SIGMOD programming contest 2021 presented the participants with an entity resolution task[4]. The goal was to deduplicate three datasets and achieve the highest average f1 score. All participants were given the opportunity to use Snowman as a pre-configured evaluation tool to investigate matching results. After the contest finished, we analyzed five high performing matching solutions with our benchmark platform on the evaluation dataset Z_4. Three of the matching solutions used a machine learning approach, one used a rule-based approach, and one used a combination of rules and machine learning. The dataset Z_4 is a dense dataset (6.2% attribute values are missing) with a small schema (4 attributes). The values of attribute *name* are significantly longer than those of the other attributes, and contain much information in unstructured form. The attributes *brand* and *size* often contain redundant information that is already included in *name*. 1.9% of all pairs are true duplicates and in average clustered in groups of 7. In the following, we present our key insights while showcasing how Snowman helped us uncover them.

For an initial overview, we used Snowman's N-Metrics Viewer (Figure 5) to compare quality metrics, such as precision, recall, and f1 score. On average, the top-5 contest teams achieved an f1 score of 90.34% with 87.4% as the minimum and 92.7% as the maximum. These results are impressive, as the dataset constitutes a quite difficult matching task: most of the matching has to be based on unstructured information in the attribute *name*.

As the performance of a matching solution is often strongly related to the selected similarity threshold, metric/metric diagrams as introduced in Section 4.5.1 can be used to find the optimal threshold. Using Snowman, we ascertained that two matching solutions had in fact not selected the optimal similarity threshold for their results. Selecting a higher similarity threshold would have increased their f1 score by 8% and 6%, respectively. Surprisingly, these observations are also true for the training dataset.

With Snowman, we identified three true duplicate pairs that were not detected by at least four solutions. This evaluation can be accomplished with the N-Intersection Viewer (see Figure 1) by subtracting all result sets from the ground truth. Interestingly, all three pairs include the record with ID *altosight.com//1420*. This is an indicator that this record is especially difficult to match.

These findings confirm that useful insights can be gained by applying structured evaluation techniques and result exploration,

[2] https://github.com/HPI-Information-Systems/snowman
[3] https://hpi-information-systems.github.io/snowman
[4] https://dbgroup.ing.unimore.it/sigmod21contest/

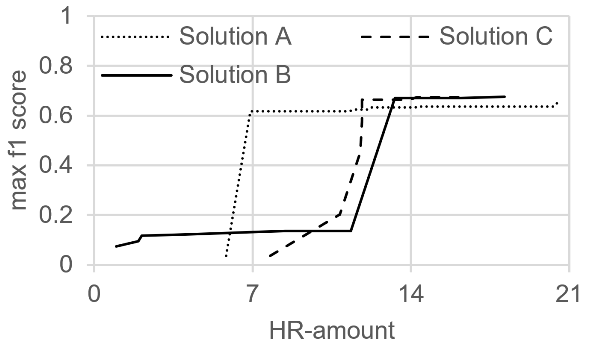


Figure 6: Maximum f1 score against effort spent (in hours). We optimized three solutions for the SIGMOD D4 dataset from scratch and tracked the effort spent throughout the process.

emphasizing the need for a benchmark platform. In Appendix C, we explore differences between the contest's test and training data. In summary, Snowman can help to better understand a matching task as well as result sets and thereby accelerates the development of successful matching solutions.

5.3 Experimental Evaluation of Soft KPIs

We conducted a study to validate what impact *effort*, as discussed in Section 3.3, has on matching performance. Our expectation was that matching solutions improve with additional effort invested into their configuration, and that the curve of a target metric (e.g., f1 score) asymptotically approaches an optimum – specific to each matching solution and dataset. To validate this expectation, we manually optimized three different matching solutions, ranging from rule-based to machine learning approaches, for a given dataset. Specifically, we deduplicated the SIGMOD contest's D4 dataset (see previous section) with the goal to optimize the f1 score achieved on the test dataset Z4 by using the training dataset X4 as well as its ground truth annotation. Throughout the process, we tracked the effort spent. Figure 6 illustrates how the f1 score evolved against the effort.

Each solution had a breakthrough point-in-time at which the performance increased significantly. Afterwards, all solutions reached a barrier at around 14 hours, above which only minor improvements were achieved. This could either mean that a major configuration change is required or that the maximum achievable performance for this matching solution on dataset D4 is reached.

Additionally, we analyzed the f1 score of the submissions from five top teams of the SIGMOD contest over time (see Figure 7). The matching quality of the different teams generally increased over time, but sometimes faced significant declines in matching performance. Thus, the matching task had an overall trial-and-error character, which indicates that the dataset D4 seems to be challenging even for matching specialists.

In conclusion, effort diagrams are beneficial in a variety of ways: They help users track the cost spent for optimizing matching solutions, to detect time points when larger configuration changes are necessary or additional effort might be wasted, and give insights about the difficulty of a dataset.

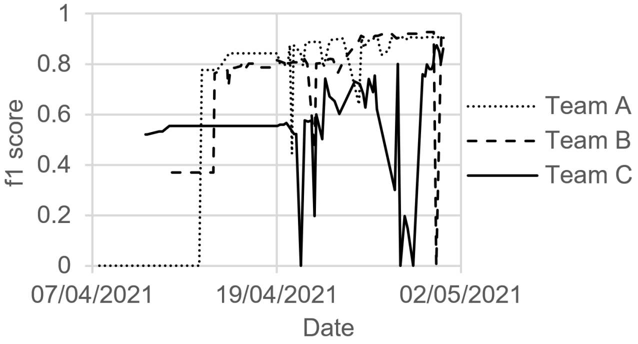


Figure 7: f1 score over time at the SIGMOD contest. This diagram illustrates the evolution of the f1 score on dataset D4 of three of the five top teams at the SIGMOD contest over time.

6 CONCLUSION AND OUTLOOK

We introduced Frost, a benchmark platform for entity resolution systems. Besides the traditional benchmark evaluation for result quality, it offers a dimension for expenditures as well as techniques to systematically explore and understand data matching results. We examined how Frost can benefit both organizations in their buying decision and developers in improving their matching solutions. Finally, we presented Snowman as a reference implementation for Frost and evaluated the results of this year's top teams from the SIGMOD programming contest with it. Although we consider Frost and Snowman a significant step in the direction of a standardized and comprehensive benchmark platform for entity resolution systems, our long-term goal is to advance Frost even further. Thus, there is still potential for future work:

- **Selecting benchmark datasets**: As discussed in Section 3.2.4, it is difficult to find representative benchmark datasets for a real-world matching task. A suitability score based on profiling metrics would be an important contribution towards the search for suitable benchmark datasets.
- **Categorizing errors**: The ability to categorize the errors of a matching solution helps to more easily find structural deficiencies. For example, a matching solutions could be especially weak in in the handling of typos.
- **Recommending matching solutions**: A long-term goal might be to gather matching solutions, benchmark datasets, and evaluation results in a central repository. To assist organizations with real-world matching tasks, Frost could use this information to automatically determine the most promising matching solutions.

REFERENCES

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where are we and what needs to be done? *VLDB Endowment* 9, 12 (2016), 993–1004. https://doi.org/10.14778/2994509.2994518

[2] Patricia C. Arocena, Boris Glavic, Giansalvatore Mecca, Renée J. Miller, Paolo Papotti, and Donatello Santoro. 2015. Messing Up with BART: Error Generation for Evaluating Data-Cleaning Algorithms. *PVLDB* 9, 2 (2015), 36–47. https://doi.org/10.14778/2850578.2850579

[3] Matt Barnes. 2015. A Practioner's Guide to Evaluating Entity Resolution Results. *Computing Research Repository (CoRR)* (2015). http://arxiv.org/abs/1509.04238

[4] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)* 57, 1 (1995), 289–300. http://www.jstor.org/stable/2346101

[5] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. 2009. Swoosh: a generic approach to entity resolution. *VLDB Journal* 18, 1 (2009), 255–276. https://doi.org/10.1007/s00778-008-0098-x

[6] Jens Bleiholder and Felix Naumann. 2008. Data fusion. *Comput. Surveys* 41, 1 (2008), 1:1–1:41. https://doi.org/10.1145/1456650.1456651

[7] Sabri Boughorbel, Fethi Jarray, and Mohammed El-Anbari. 2017. Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric. *PLOS ONE* 12, 6 (2017), e0177678. https://doi.org/10.1371/journal.pone.0177678

[8] Prodromos D Chatzoglou and Linda A Macaulay. 1997. The importance of human factors in planning the requirements capture stage of a project. *International Journal of Project Management* 15, 1 (1997), 39–53. https://doi.org/10.1016/S0263-7863(96)00038-5

[9] Davide Chicco, Niklas Tötsch, and Giuseppe Jurman. 2021. The Matthews correlation coefficient (MCC) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation. *BioData Mining* 14, 1 (2021), 13. https://doi.org/10.1186/s13040-021-00244-z

[10] Peter Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer Verlag. https://doi.org/10.1007/978-3-642-31164-2

[11] Peter Christen. 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 24, 9 (2012), 1537–1555. https://doi.org/10.1109/TKDE.2011.127

[12] Peter Christen. 2014. *Preparation of a real temporal voter data set for record linkage and duplicate detection research.* Technical Report. The Australian National University.

[13] Peter Christen and Dinusha Vatsalan. 2013. Flexible and Extensible Generation and Corruption of Personal Data. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM).* Association for Computing Machinery, 1165–1168. https://doi.org/10.1145/2505515.2507815

[14] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2019. End-to-End Entity Resolution for Big Data: A Survey. *Computing Research Repository (CoRR)* (2019). http://arxiv.org/abs/1905.06397

[15] Valter Crescenzi, Andrea De Angelis, Donatella Firmani, Maurizio Mazzei, Paolo Merialdo, Federico Piai, and Divesh Srivastava. 2021. Alaska: A Flexible Benchmark for Data Integration Tasks. *Computing Research Repository (CoRR)* (2021). https://arxiv.org/abs/2101.11259

[16] Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, Pradap Konda, Yash Govind, and Derek Paulsen. 2016. The Magellan Data Repository. https://sites.google.com/site/anhaidgroup/useful-stuff/data.

[17] Ben Davis. 2014. The cost of bad data: stats. https://econsultancy.com/the-cost-of-bad-data-stats.

[18] Dong Deng, Wenbo Tao, Ziawasch Abedjan, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2017. Entity Consolidation: The Golden Record Problem. *Computing Research Repository (CoRR)* (2017). http://arxiv.org/abs/1709.10436

[19] AnHai Doan, Alon Y. Halevy, and Zachary G. Ives. 2012. *Principles of Data Integration.* Morgan Kaufmann. http://research.cs.wisc.edu/dibook/

[20] Uwe Draisbach, Peter Christen, and Felix Naumann. 2020. Transforming Pairwise Duplicates to Entity Clusters for High-quality Duplicate Detection. *Journal on Data and Information Quality (JDIQ)* 12, 1 (2020), 3:1–3:30. https://doi.org/10.1145/3352591

[21] Uwe Draisbach and Felix Naumann. 2010. DuDe: The duplicate detection toolkit. In *Proceedings of the International Workshop on Quality in Databases (QDB).* VLDB, Singapore, Singapore.

[22] Uwe Draisbach and Felix Naumann. 2013. On choosing thresholds for duplicate detection. In *The International Conference on Information Quality (ICIQ).* MIT Information Quality (MITIQ) Program.

[23] Byron A Ellis. 2007. Life cycle cost. In *International Conference of Maintenance Societies.* Maintenance Engineering Society of Australia, 1–8.

[24] Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2014. NADEEF/ER: generic and interactive entity resolution. In *International Conference on Management of Data (SIGMOD).* ACM, 1071–1074. https://doi.org/10.1145/2588555.2594511

[25] Khaled El Emam, Lucy Mosquera, and Richard Hoptroff. 2020. *Practical Synthetic Data Generation - Balancing Privacy and the Broad Availability of Data.* O'Reilly.

[26] Roy T. Fielding and Richard N. Taylor. 2000. Principled design of the modern Web architecture. In *Proceedings of the International Conference on Software Engineering (ICSE).* Association for Computing Machinery, 407–416. https://doi.org/10.1145/337180.337228

[27] Edward B Fowlkes and Colin L Mallows. 1983. A method for comparing two hierarchical clusterings. *Journal of the American statistical association* 78, 383 (1983), 553–569. https://doi.org/10.2307/2288117

[28] Luzia Gonçalves, Ana Subtil, and M Rosário Oliveira P d Bermudez. 2014. ROC curve estimation: An overview. *REVSTAT* 12, 1 (2014), 1–20.

[29] David J. Hand and Peter Christen. 2018. A note on using the F-measure for evaluating record linkage algorithms. *Statistics and Computing* 28, 3 (2018), 539–547. https://doi.org/10.1007/s11222-017-9746-6

[30] David J. Hand, Peter Christen, and Nishadi Kirielle. 2021. F*: an interpretable transformation of the F-measure. *Mach. Learn.* 110, 3 (2021), 451–456. https://doi.org/10.1007/s10994-021-05964-1

[31] Oktie Hassanzadeh, Fei Chiang, Renée J. Miller, and Hyun Chul Lee. 2009. Framework for Evaluating Clustering Algorithms in Duplicate Detection. *PVLDB* 2, 1 (2009), 1282–1293. https://doi.org/10.14778/1687627.1687771

[32] Alireza Heidari, George Michalopoulos, Shrinu Kushagra, Ihab F. Ilyas, and Theodoros Rekatsinas. 2020. Record fusion: A learning approach. *Computing Research Repository (CoRR)* (2020). https://arxiv.org/abs/2006.10208

[33] Arvid Heise, Gjergji Kasneci, and Felix Naumann. 2014. Estimating the Number and Sizes of Fuzzy-Duplicate Clusters. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM).* Association for Computing Machinery, 959–968. https://doi.org/10.1145/2661829.2661885

[34] Al Koudous Idrissou, Frank van Harmelen, and Peter van den Besselaar. 2018. Network Metrics for Assessing the Quality of Entity Resolution Between Multiple Datasets. In *Knowledge Engineering and Knowledge Management (EKAW) (Lecture Notes in Computer Science)*, Vol. 11313. Springer Verlag, 147–162. https://doi.org/10.1007/978-3-030-03667-6_10

[35] Ekaterini Ioannou, Nataliya Rassadko, and Yannis Velegrakis. 2013. On Generating Benchmark Data for Entity Matching. *Journal on Data Semantics* 2, 1 (2013), 37–56. https://doi.org/10.1007/s13740-012-0015-8

[36] Ekaterini Ioannou and Yannis Velegrakis. 2019. EMBench^{++}: Data for a thorough benchmarking of matching-related methods. *Semantic Web* 10, 2 (2019), 435–450. https://doi.org/10.3233/SW-180331

[37] Pradap Konda, Sanjib Das, Paul Suganthan G. C., AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeffrey F. Naughton, Shishir Prasad, Ganesh Krishnan, Rohit Deep, and Vijay Raghavendra. 2016. Magellan: Toward Building Entity Matching Management Systems. *VLDB Endowment* 9, 12 (2016), 1197–1208. https://doi.org/10.14778/2994509.2994535

[38] Hanna Köpcke and Erhard Rahm. 2010. Frameworks for entity matching: A comparison. *Data and Knowledge Engineering (DKE)* 69, 2 (2010), 197–210. https://doi.org/10.1016/j.datak.2009.10.003

[39] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2009. Comparative evaluation of entity resolution approaches with FEVER. *PVLDB* 2, 2 (2009), 1574–1577. https://doi.org/10.14778/1687553.1687595

[40] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Benchmark datasets for entity resolution. https://dbs.uni-leipzig.de/research/projects/object_matching/benchmark_datasets_for_entity_resolution.

[41] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *PVLDB* 3, 1 (2010), 484–493. https://doi.org/10.14778/1920841.1920904

[42] Ioannis K. Koumarelas, Lan Jiang, and Felix Naumann. 2020. Data Preparation for Duplicate Detection. *Journal on Data and Information Quality (JDIQ)* 12, 3 (2020), 15:1–15:24. https://dl.acm.org/doi/10.1145/3377878

[43] B. Lakhanpal. 1993. Understanding the factors influencing the performance of software development groups: An exploratory group-level analysis. *Information and Software Technology* 35, 8 (1993), 468–473. https://doi.org/10.1016/0950-5849(93)90044-4

[44] Marina Meila. 2003. Comparing Clusterings by the Variation of Information. In *Annual Conference on Computational Learning Theory and Kernel Workshop (Lecture Notes in Computer Science)*, Vol. 2777. Springer Verlag, Washington, DC, USA, 173–187. https://doi.org/10.1007/978-3-540-45167-9_14

[45] David Menestrina, Steven Whang, and Hector Garcia-Molina. 2010. Evaluating Entity Resolution Results. *PVLDB* 3, 1 (2010), 208–219. https://doi.org/10.14778/1920841.1920871

[46] Charini Nanayakkara, Peter Christen, Thilina Ranbaduge, and Eilidh Garrett. 2019. Evaluation measure for group-based record linkage. *International Journal of Population Data Science (IJPDS)* 4, 1 (2019). https://doi.org/10.23889/ijpds.v4i1.1127

[47] Fabian Panse and Felix Naumann. 2021. Evaluation of Duplicate Detection Algorithms: From Quality Measures to Test Data Generation. In *Proceedings of*

the International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2373–2376. https://doi.org/10.1109/ICDE51399.2021.00269

[48] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2019. A Survey of Blocking and Filtering Techniques for Entity Resolution. *Computing Research Repository (CoRR)* (2019). http://arxiv.org/abs/1905.06167

[49] Petar Petrovski and Christian Bizer. 2020. Learning expressive linkage rules from sparse data. *Semantic Web* 11, 3 (2020), 549–567. https://doi.org/10.3233/SW-190356

[50] Anna Primpeli and Christian Bizer. 2020. Profiling Entity Matching Benchmark Tasks. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. Association for Computing Machinery, 3101–3108. https://doi.org/10.1145/3340531.3412781

[51] Erhard Rahm and Hong Hai Do. 2000. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin* 23, 4 (2000), 3–13. http://sites.computer.org/debull/A00DEC-CD.pdf

[52] M. Ali Rostami, Alieh Saeedi, Eric Peukert, and Erhard Rahm. 2018. Interactive Visualization of Large Similarity Graphs and Entity Resolution Clusters. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 690–693. https://doi.org/10.5441/002/edbt.2018.86

[53] Frank Ruskey, Carla D Savage, and Stan Wagon. 2006. The search for simple symmetric Venn diagrams. *Notices of the AMS* 53, 11 (2006), 1304–1311.

[54] Sunita Sarawagi and Anuradha Bhamidipaty. 2002. Interactive deduplication using active learning. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*. Association for Computing Machinery, 269–278. https://doi.org/10.1145/775047.775087

[55] Tzanina Saveta. 2014. *SPIMBench: A Scalable, Schema-Aware Instance Matching Benchmark for the Semantic Publishing Domain.* Master's thesis. University of Crete.

[56] Joseph T Schaefer. 1990. The critical success index as an indicator of warning skill. *Weather and forecasting* 5, 4 (1990), 570–575. https://doi.org/10.1175/1520-0434(1990)005<0570:TCSIAA>2.0.CO;2

[57] Claude E Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[58] John R Talburt. 2011. *Entity resolution and information quality.* Morgan Kaufmann.

[59] John R. Talburt, Yinle Zhou, and Savitha Yalanadu Shivaiah. 2009. SOG: A Synthetic Occupancy Generator to Support Entity Resolution Instruction and Research. In *The International Conference on Information Quality (ICIQ)*. MIT Press, 91–105.

[60] Robert E Tarjan. 1972. *On the Efficiency of a Good but Not Linear Set Union Algorithm.* Technical Report. Tech. Rep. 72-148, Dept. of Comp. Sci., Cornell University, Ithaca, NY, USA. https://doi.org/10.1145/321879.321884

[61] Tobias Vogel, Arvid Heise, Uwe Draisbach, Dustin Lange, and Felix Naumann. 2014. Reach for gold: An annealing standard to evaluate duplicate detection results. *Journal on Data and Information Quality (JDIQ)* 5, 1-2 (2014), 5:1–5:25. https://doi.org/10.1145/2629687

[62] Melanie Weis, Felix Naumann, and Franziska Brosy. 2006. A duplicate detection benchmark for XML (and relational) data. In *Proceedings of the International Workshop on Information Quality for Information Systems (IQIS)*. Association for Computing Machinery.

A SNOWMAN'S RUN-EVERYWHERE HIGH-PERFORMANCE BACK-END

Building the reference implementation Snowman[5] for Frost is a challenging task, because the benchmark platform has to meet several conflicting requirements. Our final application stack makes use of ElectronJS and splits Snowman into a NodeJS back-end and a webapp as its front-end. In this section, we outline requirements and decisions that led to Snowman's application stack, as well as discuss its benefits and shortcomings.

The following list summarizes the key requirements that influenced Snowman's development:

(1) The application should be able to run on all major operating systems, including Windows 10, macOS 11 Big Sur as well as current versions of Ubuntu and Debian.
(2) Deployment should be possible both on local computers and in shared cloud environments.

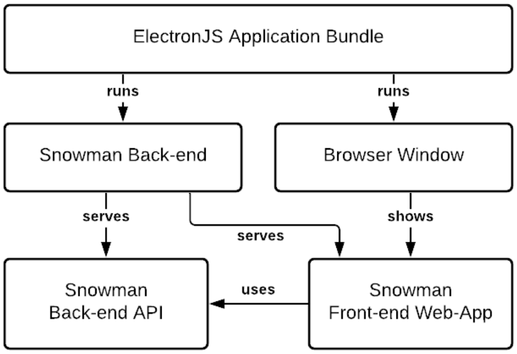


Figure 8: Snowman's Back-end Architecture.

(3) For local deployment, no installation or external dependencies should be required. Also, Snowman should be able to run without administrative privileges, as these are usually unavailable in commercial settings.
(4) Installation, upgrade, and removal procedures should be as simple as those of apps on a smartphone.
(5) Snowman should be easy to integrate with 3rd party applications via an http-based application programming interface (API).
(6) All functionality included with in the front-end should also be made available through the API.
(7) The tool should be interactive and answer usual requests in less than one second.

A.1 Architecture

Snowman's architecture is split into a front-end and a back-end, which run completely independent and communicate via an API. Section A.4 outlines Snowman's API interface in detail. As the front-end uses an API to communicate with the back-end, it does not contain functionality that cannot be also accessed via Snowman's API. See Figure 8 for an overview of Snowman's architecture.

Both front-end and back-end run on ElectronJS[6], which allows the application to be shipped as a single binary file. As Snowman's front-end is a web application, it is based on web technologies, such as HTML, CSS and JavaScript. The latter is also used in the back-end. To detect and prevent issues caused by loose typing, the TypeScript preprocessor[7] is used to enforce strict type checking. See Section A.2 for more details.

As Snowman describes a benchmark for relational data only, a relational database was chosen to persist data within the back-end. Relational databases allow for quick access to all records in different sorting orders using index structures. Normal database management systems (DBMS) require a separate process to install and operate. To mitigate the need for external dependencies, we chose SQLite[8] as our DBMS, which can be bundled together with the application.

[5] https://github.com/HPI-Information-Systems/snowman

[6] https://www.electronjs.org

[7] https://www.typescriptlang.org

[8] https://www.sqlite.org/index.html

Dataset	Record Count	Matched Pairs	Metric Diagram Custom	Metric Diagram Naïve	Approximate Speedup Factor
Altosight X4	835	4,005	184ms	1.7s	9
HPI Cora	1,879	5,067	245ms	7.4s	30
FreeDB CDs	9,763	147	293ms	16.4s	56
Songs 100k	100,000	45,801	1.6s	43.9s	28
Magellan Songs	1,000,000	144,349	6.1s	6min 43s	66

Table 1: Run-time of Metric/Metric Diagrams. The table shows a comparison of the run-time of Snowman's optimized algorithm for metric/metric diagrams against the naïve approach. Both approaches are described in Appendix D. For each Metric/Metric diagram, 100 different similarity thresholds have been calculated ($s = 100$).

A.2 Platform

ElectronJS is a toolkit to bundle web applications into standalone and multi-platform desktop apps. Each application consists of a front-end webpage that runs inside a slimmed-down Chromium browser window, as well as a back-end process running on NodeJS (see Figure 8). The chromium engine thereby isolates the web application from the host computer similar to a normal web browser. Privileged operations including access to system resources have to pass through the back-end process with inter-process communication. Accordingly, it remains to the back-end to sanitize and authorize requests received from the front-end before passing them to the operating system. After the application startup phase, Snowman does not make use of inter-process communication but rather passes all communication through the REST API.

The installation process is simple, as all program code resides within the downloaded artifact. To upgrade, one has to download a newer artifact and replace the old one. Similarly, uninstalling is handled by deleting the artifact and the application data.

By using ElectronJS, we benefit from the large ecosystem around JavaScript code. In fact, JavaScript's package manager NPM is considered the largest ecosystem for open-source libraries in the world[9]. All functionality defined within the NodeJS standard library is platform-independent. This alleviates the need to write platform-specific code and amounts to cleaner code.

A.3 Data Storage

Although SQLite3 has many advantages as outlined above, it also comes with a major drawback. Each database file needs to be locked before a write operation can occur. During a lock-phase, other requests cannot read or write the target database file. Since Snowman's back-end is single-threaded, we do not consider this as a practical limitation. Additionally, one could easily prevent this behavior by splitting the data into multiple database files through slight code changes. To access the data efficiently, we developed an Object Relational Mapping (ORM). It constructs objects out of data retrieved from the database while enforcing strict typing. Compared to other ORMs, it was created precisely for our use case and is therefore faster and more flexible. Performance improvements are mainly achieved by a specific caching algorithm for our SQL statements. Additionally, this ORM enables the back-end to dynamically create schemata for each dataset or experiment on an existing connection.

A.4 REST-ful API

As Snowman's back-end and front-end communicate through an API, it has to be well-structured and properly documented. We designed all routes according to the concept of Representational State Transfer [26] and documented them in an OpenAPI 3.0 compliant API specification[10]. Besides an easy-to-understand API, we also gain the ability to automatically generate a compatible client and server based on the API specification. Thus, correct and consistent types for our domain objects are maintained automatically across front-end and back-end. Generators[11] exist for a variety of programming languages, which allows Snowman's users to easily integrate Snowman API into their own code. For example, one could automatically upload results into a (potentially shared) Snowman instance after the code finished execution. The latest API specification can be explored interactively as part of Snowman's documentation[12].

A.5 Client-Server Version Mismatch

All client-server applications face issues when the versions of client and server mismatch. A traditional solution to this problem is to enforce versioning and have both client and server be backwards-compatible.

Instead, Snowman's client retrieves the front-end webapp from the back-end it is connected to – no matter whether it is running locally as part of the bundle or remote on a server. Thereby, the front-end presented to the user is always the same version as the back-end it is communicating with. This has the additional advantage that users can also access the remote front-end with a normal web browser.

Snowman's CLI faces a similar issue as it also makes use of Snowman API. As the Restish[13] CLI used is not specific to Snowman but rather a generic tool, it cannot face a version mismatch. Instead, it will retrieve the API specification before every request and adjust its behavior accordingly.

[9] https://mirzaleka.medium.com/exploring-javascript-ecosystem-popular-tools-frameworks-libraries-7901703ec88f

[10] https://www.openapis.org

[11] https://github.com/OpenAPITools/openapi-generator

[12] https://hpi-information-systems.github.io/snowman/openapi

[13] https://rest.sh

A.6 Performance

Besides interactive visualizations, Snowman's main goal is to analyze matching results quickly. Therefore, the back-end must allow for swift calculation of all necessary computations. Although JavaScript is not known for outstanding performance, experiments show that most workloads are not significantly slower than similar implementations in Java[14]. In its current implementation, Snowman does not implement any parallel processing. Instead, NodeJS' default single-threaded event loop is used to handle all API requests, as we would otherwise have to synchronize manually. Still, most requests can be executed in a matter of milliseconds and long-running requests usually occur only for less usual imports and exports which are not part of the actual data analysis.

As a first step to profile Snowman's performance, we measured how long rare requests take. A typical import operation for datasets with fewer than 10 000 tuples takes less than 1 second. The large Magellan Songs dataset with 1 000 000 records [16] requires about 1:15 minutes to complete its import.

As Snowman is an interactive benchmark platform, evaluation operations need to be especially fast. Thus, we also profiled Snowman's performance in the generation of metric/metric diagrams as introduced in Section 4.5.1. The exact algorithm is outlined in Appendix D.

We tested the performance with five different benchmark setups based on the Altosight X4 dataset from the SIGMOD contest 2021, the HPI Cora datset[15], the HPI FreeDB CDs dataset[16], a subset of 100 000 songs from the Magellan Songs dataset as well as the whole Magellan Songs dataset. Table 1 shows our results on the computation time required for a Metric/Metric diagram. The measurements reveal a significant speed-up factor for our custom algorithm compared to the naïve approach (see Appendix D). Even the largest dataset we tested requires only about 6.1*s* to finish the computation. Subsequent evaluations make use of caching, which yields an additional performance improvement.

All tests were conducted with Snowman version 3.2.0[17] on a typical enterprise Windows 10 laptop with an Intel i5 quad-core processor 8th gen (Hyper-Threading enabled), 16 GB of DDR4 RAM and SSD storage. For fairness, both approaches were allowed use index structures created during the initial import.

Although very large datasets require significantly more processing time than mediums-sized ones, all major algorithms used to calculate evaluation results feature less than quadratic worst-case run time in the number of records. In case the performance would become a bottleneck in the future, multiple options to cross-compile JavaScript code to native code exist. Also, computation could be split into multiple parts to enable multi-threaded processing or outsourced to a dedicated back-end such as Apache Spark.

A.7 Conclusion

In conclusion, Snowman's architectural decisions were closely dictated by the requirements. Although other frameworks exist that could offer some of these features, the environment we chose is especially flexible and reflects the state-of-the-art. With the initial open-source release in March 2021, we took the first step towards building a community on GitHub. With public documentation[18] and transparent development discussions, we hope to attract researchers and businesses world-wide to collaborate on Snowman so that it can improve and grow further.

B SNOWMAN'S FRONT-END PLATFORM ARCHITECTURE

Snowman addresses three groups of users: developers, researchers and data stewards. In particular, for data stewards, we do not assume experience in using a command line interface as provided by the Snowman back-end. Thus, Snowman features a graphical user interface. To simplify the onboarding of new users, one would ideally use fixed evaluation workflows that guide them throughout the evaluation. The apparent solution is to have an independent linear workflow for each user group and thereby assume disjunct sets of evaluations for each of them. In practice, these sets are not disjoint, but rather heavily overlap. Therefore, evaluation tools need to be independent of particular personas.

Evaluations are configured by selected benchmark components. Benchmark components determine what is to be compared; matching solutions, experiments or datasets. Furthermore, evaluation opportunities are manifold as described in Sections 2.2, 3.2, 3.3, and 4. Throughout Snowman's lifecycle, additional evaluation strategies may become relevant.

In the end, Snowman requires to be a platform for different evaluation tools that receive their configuration from a generic configurator and operate independently of each other. The platform architecture acts as a run-time for evaluation and benchmark tools called sub-apps (e.g., Metric/Metric Diagrams from Section 4.5.1), encapsulates them into separate strategies, and provides them with common features.

B.1 Front-end Technology Introduction

Snowman is based on ReactJS as it is a declarative and component-based library and thereby matches with the intuition of building a platform. Due to the popularity of ReactJS among web developers[19], it is well suited for an open-source project that depends on future community contributions. The underlying architectural design pattern is mainly determined by the state management solution chosen. Besides the native ReactJS state features, there exist alternative libraries that are more adequate for complex applications[20], such as Redux[21] and MobX[22]. Snowman uses Redux as it has great popularity and is used in many complex applications[23]. Redux is based on the *Flux* paradigm introduced by Facebook, but further expands the paradigm to make it more powerful. For instance, it describes a complex dispatcher structure which is constituted by reducers, as explained below. This additional complexity aids in the construction of complex applications such as Snowman.

[14] https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/javascript.html

[15] https://hpi.de/naumann/projects/repeatability/datasets/cora-dataset.html

[16] https://hpi.de/naumann/projects/repeatability/datasets/cd-datasets.html

[17] https://github.com/HPI-Information-Systems/snowman/releases/tag/v3.2.0

[18] https://hpi-information-systems.github.io/snowman

[19] https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted

[20] https://kentcdodds.com/blog/prop-drilling

[21] https://redux.js.org

[22] https://mobx.js.org/README.html

[23] https://www.npmjs.com/browse/depended/react-redux

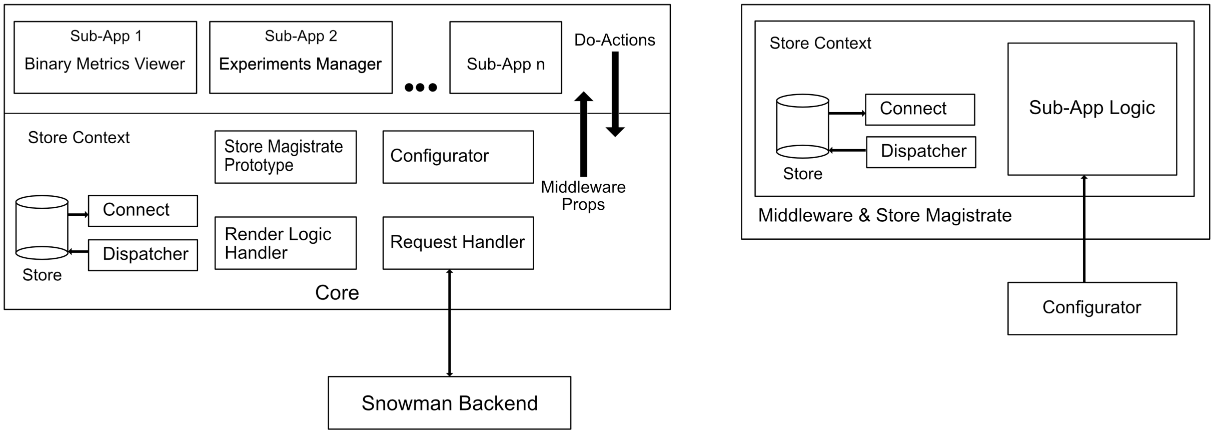


Figure 9: Snowman's Front-end Architecture (left) and the structure of a sub-app (right).

Generally, a front-end designed according to the Flux paradigm consists of view components that always reflect the state of the data. If the data changes, the view reacts to the state change.

B.2 The Platform Architecture

Comparing different experiments on a single dataset with the help of quality metrics as introduced in Section 3.2, constitutes a significant workflow. As this workflow represents a linear user story, chained reducers model it adequately: One chooses a target dataset as well as related experiments, and is then presented with applicable evaluation tools. Matching solutions thereby only serve as a filter criterion.

Additional benchmark views dock easily on to this linear workflow, but are limited to evaluate experiments of a single dataset. Thus, more complex resource selections, such as experiments from multiple datasets, are prohibited by this concept.

Therefore, adding further user stories requires the implementation of multiple, co-existing linear workflows. However, missing data exchange between workflows would require users to configure their benchmark task for each workflow separately and thus multiple times. Additionally, as steps of different user stories cannot be combined in an obvious way, synchronized state may confuse users when switching between workflows.

As pure Redux demands a single store that contains the entire application state, further architectural issues arise: Each React component receives and accesses the entire application state. Thus, each component has to know an exhaustive set of details about the store. Beyond that, it also has to register new concerns at the root reducer.

To put it in a nutshell, a pure Redux architecture featuring linear workflows does not meet the platform requirements outlined above.

As an alternative to the linear workflow, we introduce a new conceptual schema that allows a platform architecture. As outlined in Figure 9, the front-end consists of a common core as well as individual sub-apps. The core provides low-level logic to all sub-apps. For instance, it handles which sub-app is currently active and provides the http request handler. Each sub-app represents a benchmark or evaluation tool as well as tools to manage datasets, experiments and matching solutions.

A generic benchmark configurator is part of the core, too. It is able to model all possible configuration workflows: Based on the currently selected evaluation tool, adequate benchmark components can be selected. Limitations are defined by the evaluation tool itself. In contrast to the linear workflow, all required interpretations (i.e., which experiment is the ground truth) are thereby encoded within the configurator.

As part of our Redux architecture, we opted for each sub-app, as well as the core itself, to use separate Redux stores and only allow communication over a well-defined interface. Since we thereby limit the amount of knowledge required by each sub-app to a minimum, we enable a true platform architecture within Snowman's front-end.

B.3 Technical Implementation

Technically, a Redux store and its associated functionality work by extending ReactJS context methods. Context methods link values and methods defined outside into the local component scope. Since it is possible to nest contexts, a component can only access resources from the context it is contained in, but not any overlying context. Thus, nesting sub-apps by context encapsulates them and only gives them access to their own store defined within the current context.

In Snowman, a self-designed middleware abstracts the encapsulation process and declares a common interface between the core and the sub-apps. Its responsibilities also include to carry out instructions from the core, for example to show the sub-app when it is active. Moreover, following the dependency injection pattern, control over store references has to be further abstracted and separated from the middleware. Additionally, as initialized Redux stores exist over the application's run-time and are not affected by garbage collection, Snowman requires a mechanism to keep track of its stores.

To address this issue, so-called store magistrates control pools of store instances. Once the sub-app is about to appear, the middleware generates an identifier and requests a store instance from the corresponding magistrate with it. If the supplied identifier already exists within the pool, the magistrate reuses the existing store instance. Otherwise, the store magistrate constructs a new store instance, pushes it to the pool, and returns its reference.

As a further optimization, we propose virtual stores, which represent slices of a materialized store and do not directly depend on a context. Thus, magistrates can control them across their whole lifecycle, including destruction. We created a reference implementation of virtual stores[24] that shows their greater flexibility.

B.4 Discussion

The platform architecture introduced above has both advantages and disadvantages. In the following, we argue in favor and against Snowman's platform architecture and its implementation.

As outlined, the platform architecture comes with a two-sided benchmark configurator. On the one hand, it encapsulates the concerns for selecting benchmark parts. Thus, sub-apps do not have to take care of these concerns themselves. Furthermore, additional configurators are not required because a single generic one can fit all possible workflows. On the other hand, the configurator incorporates an inherent complexity for developers and is therefore more complicated to maintain and extend. Additionally, we observed that end users might also be overwhelmed by its complexity and the missing guidance. Users require less support in understanding a linear, specific workflow compared to the general, standardized workflow. A UX expert confirmed these observations. To counteract these effects, a guided introduction to the most important use cases of Snowman should be added.

Next, the de-structuring into sub-apps does not only yield advantages, but might also be problematic as most sub-apps still require Snowman as their run-time environment. Thus, they cannot be used independently or would require substantial modifications to become independent.

Additionally, due to the inherent complexity and cohesion of the core, extracting a subset of its components and functionalities is difficult. Furthermore, extensions and modifications to the core require in-depth knowledge about its structure as well as the component's choreographies. For Snowman we do not consider the outlined disadvantages significant enough. Instead, advantages such as the core's small, easy to use core interface, and the fact that it is designed analogously to well-known and proven design concepts of operating systems, weigh more significant.

In conclusion, we therefore consider Snowman's platform-grade architecture to be a success. Nevertheless, it has to tackle the same issues as every platform-grade architecture: Changes to the interface between core and sub-apps are critical, because every sub-app has to implement the new core interface upon the change. Even though changes to the core itself are difficult, the abstraction becomes a game changer when new benchmark and evaluation apps are designed as they are truly independent of existing sub-apps. We hope that Snowman gains additional popularity through the resulting extensibility and customizability.

C CORRELATING DATASET PROPERTIES AND DATA MATCHING PERFORMANCE

The quality of a matching solution strongly depends on the dataset on which it is executed. It might perform differently on a sparse dataset than on a dense dataset, for example. A dataset with many non-atomic attributes presents a matching solution different challenges than a dataset with mainly atomic attributes. Therefore, knowing the characteristics of the underlying dataset is important to develop high-performing matching solutions.

Frost can be used to investigate differences in matching behavior when running a matching solution on datasets with different properties. An essential problem is that real-world scenarios usually do not feature ground truth annotations, which are necessary to evaluate matching solutions. However, over the past few decades, the data matching community has compiled a set of annotated datasets that are typically used by researchers to develop and evaluate their matching solutions. Those datasets are called "benchmark datasets". As described in Section 3.1.2, some of these datasets are from real-world scenarios and have been annotated through laborious processes. In contrast to researchers, practitioners usually do not want to evaluate the behavior of a newly developed matching solution, but must use these solutions to detect all duplicates within their use case-specific datasets. Since these datasets do not provide a ground truth, a typical challenge for them is to find a benchmark dataset that is similar to their given use case dataset and thus can be used for evaluation instead. This in turn requires the development of similarity metrics suitable to compare these datasets appropriately.

In the following, we analyze how different dataset properties influence the quality of a matching solution's result. For this, we evaluate the matching results on datasets they were trained on and others they were not trained on. Then we evaluate whether correlations between the performance of the data matching solutions and certain dataset characteristics exist. To do so, we use matching solutions of participants of the ACM SIGMOD programming contest 2021. The contest consists of three datasets (D_2, D_3, and D_4), which each feature a training dataset (X_2, X_3 and X_4) and a test dataset (Z_2, Z_3 and Z_4) where $Z_i = D_i \setminus X_i$. The contest participants developed a matching solution for each dataset, based on the training data. The evaluation data was not provided to the contest participants and was used to assess the quality of the submitted matching solutions. In our experiment, we simulated a real-world scenario with use case datasets which do not contain a ground truth by executing the data matching solutions of the SIGMOD programming contest on one of the datasets that was not used for their development. As D_2 and D_3 share the same schema, we executed matching solutions developed for D_2 on D_3 and vice versa.

C.1 Dataset Profiling

First, we profile the given datasets D_2 (X_2 & Z_2) and D_3 (X_3 & Z_3) using the following metrics.

- **Sparsity (SP)**: Sparsity is described by Primpeli and Bizer [50] as the relationship of missing attribute values to all attribute values of the relevant attributes. Missing attribute values are challenging for matching solutions and might cause errors [49].
- **Textuality (TX)**: Textuality is the average amount of words in attribute values [50]. Non-atomic attributes can complicate the matching task: The matching solution needs to handle long values by tokenizing them during preprocessing or by applying specific similarity measures.

[24] https://github.com/HPI-Information-Systems/snowman/pull/185

- **Tuple count (TC)**: Draisbach and Naumann showed that dataset size has influence on the optimal similarity threshold [22]. Thus, using a benchmark dataset with similar size compared to the use case dataset is preferable. Due to the laborious annotation process, existing benchmark datasets are usally small. Draisbach and Naumann further described that, for small datasets, an interval exists for choosing the best similarity threshold. With an increasing number of records, the interval becomes more narrow. Therefore, this fact should be taken into account when selecting a similarity threshold based on a benchmark dataset.
- **Positive ratio (PR)**: The positive ratio describes the relationship of the number of true duplicate pairs compared to the number of all pairs. If the number of duplicates (or at least an estimation) is known, the matching solution could use this ratio to dynamically adapt its matching decisions.
- **Vocabulary similarity (VS)**: Vocabulary similarity quantifies the similarity of the vocabularies of two datasets. Similar vocabularies might cause similar behavior of the matching solution. We calculate the vocabulary similarity using the Jaccard coefficient: Let D_1, D_2 be datasets and let $vocab(D_i)$ be the vocabulary-set of D_i, tokenized by spaces.

$$VS(D_1, D_2) = \frac{|vocab(D_1) \cap vocab(D_2)|}{|vocab(D_1) \cup vocab(D_2)|}$$

Table 2 shows that the datasets have different characteristics and therefore represent different challenges for a matching solution. Both datasets represent the same domain – notebook specifics. D_3 is a lot sparser than D_2. Both have a high textuality, but D_2 has a much higher textuality (in average 25.84). In the following, we point out the key indicators of the datasets:

- **Notebook** (D_2): The training dataset as well as the test dataset have a high textuality of 27.99 and 23.69. A possible matching solution needs to deal with this by, for example, tokenizing the attribute values. The vocabulary of X_2 and Z_2 partly overlaps. Approximately every third token exists in both datasets.
- **Notebook large** (D_3): X_3 as well as Z_3 are sparse – approximately every second attribute is missing (50.1% & 42.6%). Furthermore, the datasets include much textual data with an average attribute length of 15 words. The training and test dataset of D3 differs in its characteristics. 2.2% of all possible pairs in the training dataset X_3 are duplicates. In contrast, 12.1% of all pairs in the test dataset Z_3 are duplicates. This gap might influence the quality performance of a matching solution.

C.2 Influence of Dataset Characteristics on Matching Solutions

In the following, we analyze the influence of specific dataset characteristics on the average quality of matching solutions. We executed three matching solutions of the SIGMOD contest on X_2, X_3, Z_2 and Z_3 and loaded the results into Snowman. Snowman supports this evaluation by providing an easy way to determine the quality metrics of experiments and to choose the best similarity threshold for each matching solution. In the next part, we always refer to the average quality metrics over these solutions.

	D_2 dataset		D_3 dataset	
Dataset	Train (X_2)	Test (Z_2)	Train (X_3)	Test (Z_3)
SP	11.1%	19.72%	50.1%	42.6%
TX	27.99	23.69	15.53	15.35
TC	58'653	18'915	56'616	35'778
PR	2.2%	3.6%	2.2%	12.1%
VS	59.0%		37.7%	

Table 2: Profiling the datasets of the ACM SIGMOD programming contest.

As shown in Table 3, it becomes clear that matching solutions generally perform better on the datasets on which they have been developed than on new data, as expected. Taking a look at the average f1 score, one can observe that the matching solution developed for D_3 performs a lot better on D_2 (average f1 score 80.5%) than the matching solution developed for D_2 does on D_3 (average f1 score 41.4%). As outlined above, the D_3 dataset is a sparse dataset with in average 46.4% missing attribute values. The matching solutions trained on a sparse dataset performed better on a non-sparse dataset than the matching solutions that were developed on a non-sparse dataset applied to a sparse dataset. Therefore, one reason for the poor performance might be the missing data. Thus, when selecting a benchmark dataset, the sparsity should be similar to the real-world dataset or lower.

Additionally, one can observe a gap between the average quality metrics of the test- and training dataset of D_3. The test dataset Z_3 has an average f1 score of 35.7% and the training dataset X_3 has an average f1 score of 47.0%, resulting in a difference of Δf1 score = 11.3%. The difference in the average f1 score is much smaller for X_2 and Z_2: X_2 has an average f1 score of 81.3% and Z_2 has an average f1 score of 79.6%. This results in a difference of Δf1 score = 1.7%. One reason for this might be the lower vocabulary similarity of X_3 and Z_3. The vocabularies of the datasets X_2 and Z_2 are a lot more similar (59.0%) than the vocabularies of X_3 and Z_3 (37.7%). It might be easier for the classification model, developed on X_2, to classify the candidate pairs correctly, as its vocabulary is more similar to the vocabulary used during training. In summary, vocabulary similarity might be a relevant indicator for the suitability of benchmark datasets.

C.3 Conclusion

The characteristics of a dataset have a traceable influence on the performance of a matching solution. Thus, when choosing a benchmark dataset, the suitability of the properties of the benchmark dataset should be considered. Concretely, a benchmark dataset should present a matching solution with similar challenges as the use case dataset under consideration. If the datasets differ too much in their properties, the matching results are not representative.

A next step is to confirm the findings on more datasets and matching solutions. Finally, a similarity measure between datasets for the selection of a suitable benchmark dataset based on the examined dataset properties needs to be developed.

Matching solution	Metric	D_2 dataset		D_3 dataset	
		Train	Test	Train	Test
developed on X_2	Precision	100%	97.7%	46.9%	90.1%
	Recall	99.6%	97.0%	56.2%	43.2%
	f1 score	99.8%	97.4%	35.7%	47.0%
developed on X_3	Precision	76.3%	68.5%	69.7%	98.6%
	Recall	89.5%	95.0%	97.2%	97.5%
	f1 score	81.3%	79.6%	76.5%	98.2%

Table 3: Average quality metrics of matching solutions of the ACM SIGMOD programming contest. The first row contains average results of matching solutions that were developed on dataset X_2. The second row contains average results of matching solutions that were developed on dataset X_3.

D EFFICIENTLY CALCULATING METRIC/METRIC DIAGRAMS

Many matching solutions use a similarity threshold to distinguish duplicates from non-duplicates: A pair is matched if and only if its similarity score is greater than or equal to the threshold (see Section 1.2). Therefore, the similarity threshold has a large impact on matching quality. To assist users in finding good similarity thresholds, Frost includes metric/metric diagrams, such as precision/recall or f1 score/similarity diagrams (see Section 4.5.1). The data points of these diagrams correspond to pair-based quality metrics of the matching solution based on different similarity thresholds. Thus, they give the user an overview about the general performance of the matching solution and about which similarity thresholds and similarity functions work well.

As pair-based metrics can be calculated in constant time from a confusion matrix, an algorithm for calculating metric/metric diagrams can simply output a list of confusion matrices corresponding to the requested similarity thresholds. Thus, a naïve approach to calculating metric/metric diagrams is to go through the list of matches and track all sets of pairs in the confusion matrix. A problem with this approach is that to align with Snowman's concept of experiments, the matches at each step need to be transitively closed. Thus, the total number of pairs is quadratic, which means that the algorithm has quadratic run-time in the size of the dataset. A slightly more advanced (but still naïve) approach could utilize the fact that the confusion matrix of an experiment and ground truth annotations can be calculated in linear run-time by representing experiment and ground truth as clusterings and calculating the intersection between those clusterings. It could then calculate the experiment clustering, intersection, and confusion matrix newly for every requested similarity threshold. But again, while drawing metric/metric diagrams with this approach is no problem for small datasets, the run-time increases rapidly as dataset size increases (see Table 1 on page 15). This comes from the fact that the run-time is linear in the *product* of the number of requested thresholds and dataset length in both worst and best case. This makes this algorithm unsuitable for large datasets. However, this functionality can be optimized by reusing intermediate results. While this is simple for the experiment clustering, updating the intersection clustering is a challenging task because matches do not necessarily have a direct impact on the intersection clustering, but still can cause side effects later. An example is depicted in Figure 10: Consider the ground truth clustering $\{\{a,b\},\{c\}\}$ and the matches $\{b,c\}$ and $\{a,c\}$. The intersection does not change after merging $\{b,c\}$ because b and c are in different ground truth clusters. The same holds true for $\{a,c\}$, but because b and c already have been merged together before, the intersection now contains the cluster $\{a,b\}$.

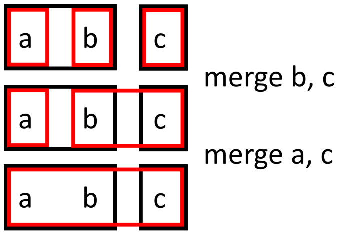


Figure 10: Intersecting two Clusterings. ground truth clustering: black, experiment clustering: red, intersection: black and red

Here, we discuss a data structure which efficiently solves this issue by explaining the algorithm used by Snowman to calculate a sequence of confusion matrices belonging to different similarity thresholds. In short, it constructs a clustering of the experiment and a clustering of the intersection between experiment and ground truth. For each requested similarity threshold, it reuses the clustering and intersection from the last requested threshold. Meanwhile, it tracks the count of pairs in experiment and intersection clustering, which allows it to efficiently calculate the confusion matrix for each requested threshold. Figure 11 illustrates the process of creating experiment and intersection clustering. The example is explained step by step throughout the section.

D.1 Input and Output

For clarity, we capitalize and pluralize collections (*Numbers*) and lowercase and singularize everything else (*number*). Clusterings are denoted as $C_{<identifier>}$. We denote the number of elements of a collection with $|\cdot|$. The algorithm receives the following parameters as input:

- D: the dataset that the matching solution was executed on
- *Matches*: array of matches predicted by the matching solution. Every entry of the array contains a similarity score and the two merged records.
- C_{truth}: ground truth duplicate clustering of D, assigning a cluster ID to every record
- s: the number of similarity scores at which metrics should be sampled. For simplicity, we assume that $|Matches|$ is divisible by $s-1$.

As output, the algorithm produces a list *Matrices* containing s confusion matrices. The naïve way to sample *Matrices* is by increasing the similarity threshold by a constant amount every time. But this has the flaw that imbalances in the distribution of thresholds can lead to data points which have a lot of matches between them and data points which have no matches between them. To combat this, the matrices are created by increasing the number of matches between two matrices by a constant amount. For example, if the similarity scores 0, 1, 3, and 6 are present and 3 matrices should be calculated, *Matrices* contains matrices for the similarity thresholds *infinity* (no matches), 3, and 0. By this, between the thresholds *infinity* and 3 lie the matches with the scores 6 and 3 and between the thresholds 3 and 0 lie the matches with the scores 1 and 0. Formally, the i-th entry of *Matrices* contains the confusion matrix of the matching solution based on the similarity threshold of the entry in *Matches* that has the $(i \cdot \frac{|Matches|}{s-1})$ biggest similarity score. If i is zero, *Matches* contains the confusion matrix of the matching solution with similarity threshold infinity.

To efficiently update the intersection clustering, we need a mechanism to know which clusters of the experiment clustering were merged between the current and the last threshold. For brevity, we assume that the union-find data structure [60] provides a mechanism for this by supporting the operation *trackedUnion*. *trackedUnion* works like a batched *union*, which outputs a list of which clusters were merged. The operation takes a list *Pairs* of record pairs as input. It calls union for the two clusters of every pair generating a new cluster ID for the resulting cluster. Then, it outputs a list *Merges*, which contains an entry for every newly created cluster that has not already been merged with another cluster. Every entry stores the ID of the newly created cluster *target* and a list *Sources* of cluster IDs from before the update, which are now part of *target*.

As an example, let the clustering contain the three clusters $\{\{a\}, \{b\}, \{c, d\}\}$ and *Pairs* contain the record pairs $\{a, b\}$ and $\{b, c\}$. After executing *trackedUnion*, the clustering would contain the cluster $\{\{a, b, c, d\}\}$ and *trackedUnion* would return *Merges* that contain exactly one entry. The entry would be composed of the IDs of the clusters $\{a\}$, $\{b\}$, and $\{c, d\}$ as *Sources* and the ID of the cluster $\{a, b, c, d\}$ as *target*. An example of *trackedUnion* for multiple consecutive invocations can be seen in Figure 11 (*trackedUnion* is executed on C_{exp}).

We also need a method to calculate the number of pairs in experiment clustering and intersection clustering efficiently. For this, we assume that the union-find data structure can track the number of pairs in each of its clusters and overall.

D.2 Algorithm Description

Algorithm 1 depicts the algorithm used by Snowman to calculate the list of confusion matrices described above. First, it creates the initial state of the experiment clustering C_{exp} using a union-find data structure with a cluster of size one for every record of dataset D. Then, it initializes the intersection clustering $C_{intersect}$ of experiment clustering C_{exp} and ground truth clustering C_{truth} as described in Section D.3. Next, the algorithm sets the output list *Matrices* to an empty list and adds the confusion matrix for the initial state of C_{exp}, C_{truth}, and $C_{intersect}$. Now, it sorts *Matches* by similarity score in descending order and splits the result into $s - 1$ ranges.

For each range, first it updates the experiment clustering C_{exp} with the *trackedUnion* function, saving its result into the list *Merges*. Then, it updates the dynamic intersection $C_{intersect}$ with the help of *Merges* as described in Section D.3. Finally, the algorithm adds the confusion matrix for the current state of C_{exp}, C_{truth}, and $C_{intersect}$ to the output list *Matrices*. After the loop terminates, *Matrices* contains s confusion matrices.

D.3 Dynamically Constructing the Intersection

The intersection clustering $C_{intersect}$ is stored as a pair of two variables: A union-find data structure for calculating the number of pairs and a map for speeding up the construction of the intersection. Each intersection cluster in the union-find data structure is uniquely identified by an experiment cluster and a ground truth cluster, and contains all records they have in common. The map uses this property as follows: The map contains an entry for every cluster of the experiment clustering e mapping to yet another map from every involved ground truth cluster g to the intersection cluster of e and g. For an example see Figure 11 column $C_{intersect}$.

Initially, the experiment clustering has a cluster of size one for every record in D (see Figure 11 row one column C_{exp}). Therefore, the initial state of the $C_{intersect}$ union-find data structure contains a cluster of size one for every record in D. Hence, the initial state of the $C_{intersect}$ map maps from every initial experiment cluster $\{r\}$ to a map containing exactly one *key* → *value* pair, namely *ground truth cluster of r* → *intersection cluster of r* (see Figure 11 row one column $C_{intersect}$).

To update $C_{intersect}$ with a list of merged clusters as returned by *trackedUnion*, Algorithm 2 iterates over each pair of *Sources* clusters and *target* cluster. First, it aggregates all intersection clusters belonging to the *Sources* clusters into a list of involved intersection clusters. Now, it groups the list by ground truth cluster into a map from ground truth cluster to intersection clusters. Then, it loops over all *ground truth cluster* → *intersection clusters* pairs and merges the intersection clusters into a new intersection cluster (the merge happens in the union-find data structure). Finally, the algorithm stores the newly created intersection clusters into a map from ground truth cluster to new intersection cluster (also contains old clusters if they were in a list of size one) and saves it into the map of $C_{intersect}$ at the position of the *target* cluster.

For an example update process, see the listing below. The example shows the process of merging a and b from Figure 11 (there is only one pair of *Sources* clusters and *target* cluster):

- **List of involved intersection clusters**: *e4g0* (*a*), *e4g1* (*c*), *e5g0* (*b*), *e5g1* (*d*)
- **Map from ground truth cluster to intersection clusters**: *g0*: *e4g0*, *e5g0*; *g1*: *e4g1*, *e5g1*
- **Merge the intersection clusters**: *e4g0*, *e5g0* → *e6g0*; *e4g1*, *e5g1* → *e6g1*
- **Map from ground truth cluster to new intersection cluster**: *g0*: *e6g0*; *g1*: *e6g1*

D.4 Exemplary Run of the Algorithm

Let the deduplicated dataset D contain the four entries a, b, c, and d. Let the ground truth clustering C_{truth} contain the two clusters $g0 : \{a, b\}$ and $g1 : \{c, d\}$ (*g0* and *g1* are the IDs of the clusters). Let the detected matches *Matches* of the matching solution be $\{a, c\}$, $\{b, d\}$, and $\{a, b\}$. Let s be 4. That means a confusion matrix should be calculated after merging each pair (and before merging the first).

Figure 11 shows the experiment clustering C_{exp}, the result of *trackedUnion*, the intersection clustering $C_{intersect}$, and the resulting confusion matrix after merging each pair (and before merging the first). Hence, the output of the algorithm contains a sequence of all depicted confusion matrices. For example, the second row (step

Step	Merge	C_{exp}	$trackedUnion(...)$	$C_{intersect}$	Confusion Matrix (only the numbers are stored)	
0		*e0* a *e1* b *e2* c *e3* d		*e0* *g0*: a *e1* *g0*: b *e2* *g1*: c *e3* *g1*: d	TP: 0 ({}) FN: 2 ({{a, b}, {c, d}})	FP: 0 ({}) TN: 4 ({{a, c}, {a, d}, {b, c}, {b, d}})
1	{a, c}	*e1* b *e3* d *e4* a, c	*src* *e0, e2* *tgt* *e4*	*e1* *g0*: b *e3* *g1*: d *e4* *g0*: a *g1*: c	TP: 0 ({}) FN: 2 ({{a, b}, {c, d}})	FP: 1 ({{a, c}}) TN: 3 ({{a, d}, {b, c}, {b, d}})
2	{b, d}	*e4* a, c *e5* b, d	*src* *e1, e3* *tgt* *e5*	*e4* *g0*: a *g1*: c *e5* *g0*: b *g1*: d	TP: 0 ({}) FN: 2 ({{a, b}, {c, d}})	FP: 2 ({{a, c}, {b, d}}) TN: 2 ({{a, d}, {b, c}})
3	{a, b}	*e6* a, b, c, d	*src* *e4, e5* *tgt* *e6*	*e6* *g0*: a, b *g1*: c, d	TP: 2 ({{a, b}, {c, d}}) FN: 0 ({})	FP: 4 ({{a, c}, {a, d}, {b, c}, {b, d}}) TN: 0 ({})

Figure 11: Exemplary run of the algorithm. Example for dataset {a, b, c, d}, ground truth clustering *g0*: {a, b}, *g1*: {c, d}, and detected matches {a, c}, {b, d}, {a, b}. The example run is described in detail in Section D.4.

Algorithm 1 Compute Confusion Matrices

```
C_exp ← new UnionFind with $|D|$ clusters of size 1
C_intersect ← initial intersection clustering (see Section D.3)
Matrices ← new list
Matrices.append(getConfusionMatrix($C_{exp}$, $C_{truth}$, $C_{intersect}$))
sort Matches by similarity score in descending order
for i from 1 to $s - 1$ do
    start ← $(i - 1) \cdot \frac{|Matches|}{s-1}$
    stop ← $i \cdot \frac{|Matches|}{s-1}$
    Pairs ← Matches[start : stop − 1]
    Merges ← $C_{exp}$.trackedUnion(Pairs)
    update $C_{intersect}$ with Merges as described in Algorithm 2
    Matrices.append(getConfusionMatrix($C_{exp}$, $C_{truth}$, $C_{intersect}$))
end for
return Matrices
```

Algorithm 2 Update Dynamic Intersection

```
for (Sources, target) in Merges do
    intersectionClusters ← empty list
    for source in Sources do
        intersectionClusters.appendAll($C_{intersect}$[source].values())
    end for
    group intersectionClusters by ground truth cluster
    $C_{intersect}$[target] ← empty map
    for (truthCluster,
            intersectionClusterGroup) in intersectionClusters do
        newCluster ← $C_{intersect}$.unionAll(intersectionClusterGroup)
        $C_{intersect}$[target][truthCluster] ← newCluster
    end for
end for
```

1) contains the state after a and c have been merged. Concretely, the experiment clustering contains the three clusters {b}, {d}, and {a, c} with the cluster IDs *e1*, *e3*, and *e4*. Because the records a and c have been merged, their respective clusters $e0$ and $e2$ from the initial experiment clustering are listed as the *source* clusters of *trackedUnion*. Because the cluster which contains a and c now has the id *e4*, the *target* cluster is *e4*. The intersection clustering contains the four clusters {b}, {d}, {a}, and {c}. In the table, every intersection cluster is shown to the right of the experiment cluster ID and ground truth cluster ID whose intersection it represents. For example, the intersection cluster {a} represents the intersection of the experiment cluster *e4* and the ground truth cluster *g0*. Therefore, it is shown to the right of *e4* and *g0*. Note that this exactly depicts the map data structure of the intersection clustering. The confusion matrix has no true positives, one false positive, two false negatives, and three true negatives. Observe that the number of true positives equals the number of pairs in $C_{intersect}$.

D.5 Conclusion

In summary, we presented Snowman's optimized algorithm for computing metric/metric diagrams. For that, we presented an operation that dynamically constructs the intersection of two clusterings. A run-time analysis of the algorithm shows that the worst-case run-time of the algorithm is in $O(|D| + |Matches| \cdot (s + log(|Matches|))$. When the ground truth has fewer than $|D|$ clusters, the complexity is even better (if it has $O(1)$ clusters, the run-time is in $O(|D| + |Matches| \cdot log(|Matches|))$. Additionally, the algorithm runs the faster, the more similar ground truth and experiment clusterings are. Note, that Snowman optimizes experiments when they are uploaded, which means that users of Snowman will experience a run-time between $O(|D|)$ and $O(|D| + |Matches| \cdot s)$, and *Matches* only contains pairs that cannot be inferred by transitively closing. Table 1 confirms that the algorithm is considerably faster compared to the naïve approach.

An interesting extension to metric/metric diagrams is a timeline feature in which new true positives and false positives between two similarity thresholds are shown (note that Frost's and Snowman's set-based comparisons already allow this). While the concepts of the above algorithm optimize this use case to some extent, the dynamic intersection and union find data structure lack the functionality to “revert” merges: whenever the user selects a similarity threshold range starting before the end of the previous range, $O(|D|)$ time is necessary to reset the clusterings. This makes interactively exploring the timeline slow, especially for large datasets. Therefore, a useful next step is to develop an algorithm for efficiently reverting merges in the dynamic intersection and union find data structure.