Retrieve, Merge, Predict: Augmenting Tables with Data Lakes (Experiment, Analysis & Benchmark Paper)

Riccardo Cappuzzo, Gaël Varoquaux, Aimee Coelho, Paolo Papotti

Retrieve, Merge, Predict: Augmenting Tables with Data Lakes (Experiment, Analysis & Benchmark Paper)

Riccardo Cappuzzo
SODA Team - Inria Saclay
Paris, France
riccardo.cappuzzo@inria.fr

Gael Varoquaux
SODA Team - Inria Saclay
Paris, France
gael.varoquaux@inria.fr

Aimee Coelho
Dataiku
Paris, France
aimee.coelho@dataiku.com

Paolo Papotti
EURECOM
Biot, France
papotti@eurecom.fr

ABSTRACT

We present an in-depth analysis of data discovery in data lakes, focusing on table augmentation for given machine learning tasks. We analyze alternative methods used in the three main steps: retrieving joinable tables, merging information, and predicting with the resultant table. As data lakes, the paper uses YADL (Yet Another Data Lake) – a novel dataset we developed as a tool for benchmarking this data discovery task – and Open Data US, a well-referenced real data lake. Through systematic exploration on both lakes, our study outlines the importance of accurately retrieving join candidates and the efficiency of simple merging methods. We report new insights on the benefits of existing solutions and on their limitations, aiming at guiding future research in this space.

PVLDB Reference Format:
Riccardo Cappuzzo, Gael Varoquaux, Aimee Coelho, and Paolo Papotti. Retrieve, Merge, Predict: Augmenting Tables with Data Lakes (Experiment, Analysis & Benchmark Paper). PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rcap107/benchmark-join-suggestions.

1 INTRODUCTION

In our data-rich era, data lakes –large, loosely structured corpora of raw data– have emerged as central resources in both academia and industry [32]. Within data lakes, data discovery is crucial, particularly through the integration of multiple tables. This process has gained significant attention [11]. In this work, we focus on a specific usage of data lakes that requires capturing only a fraction of the tables: augmenting a base table for a machine learning (ML) task, such as in the following example scenario.

Alice, a data scientist, wants to predict the box office revenue of a movie: she has access to a table that contains information about movies (e.g., year of release, director, language, budget, ...). While she can use the available data to train a model, finding more information on the subject would be beneficial; for example, joining the table about movies with a table on the cast of those movies might help, as some actors tend to appear in movies with higher revenue. Alice has access to some large repository of data, or to some search engine that she can query [3, 17] to find additional tables. Her objective is therefore to find the tables that are most relevant to her task, to merge them with the original table and finally to use the improved table to build a model that predicts the box office revenue.

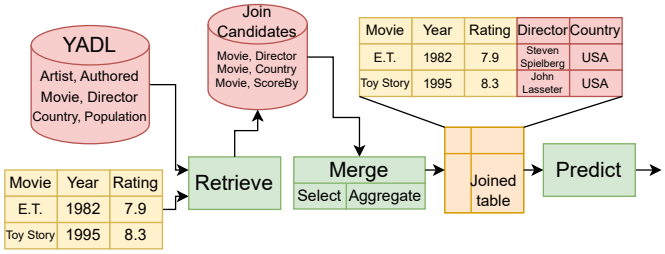

Figure 1: The pipeline. Given a base table, the three main steps (Retrieve, Merge, Predict) augment it with the information from the lake to improve the prediction performance.

As depicted in Figure 1, research on this problem is scattered across three main steps that involve four tasks: (1) **retrieving** the tables that are joinable with the original table based on shared attributes [3, 9, 12, 13, 42, 44], (2) **merging** the information by executing the joins that most improve the performance of the subsequent ML model [6, 10, 15, 28], (3) **aggregating** results in cases of one-to-many or many-to-many joins [4, 23], (4) **predicting** with the model on the resulting table [18, 37]. For each task a variety of solutions have been proposed, yet each method is evaluated within disparate datasets and for the specific task it solves, rather than as a component of the full pipeline. In the absence of a unified benchmark to compare and evaluate these methods, it becomes hard for users like Alice to select the most suitable combination of methods for their specific use case.

Our study aims to promote a deeper understanding of discovery within data lakes for an analytic task. Our dataset, YADL (Yet

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Another Data Lake) makes such analysis reproducible and extensible. We leverage YADL to systematically explore the underlying mechanisms of end-to-end learning pipelines within data lakes.

YADL is constructed on the YAGO knowledge base [29] to serve as a controlled environment for testing various methods by providing a defined vocabulary that includes entity names, thus alleviating issues stemming from uncertain matches and other ambiguities. In addition, it can be instantiated with tables in long or wide form, to test the robustness of the discovery and assembly processes with different table types.

We investigate the impact of the main steps in the end-to-end process of learning from data lakes: generating potential augmentation candidates, integrating them with the base table, and subsequently evaluating the effectiveness of this assembled pipeline.

This study has two main objectives: firstly, **we aim to determine which tasks of the pipeline affect the prediction performance the most**; secondly, we develop an equitable playing field that facilitates **a holistic and comparative assessment of alternative solutions**.

The architecture of our augmentation pipeline enables the analysis of various methods in the four tasks, while also providing a range of datasets to enable a comprehensive study. Our findings suggest that the retrieval of accurate join candidates is the most important task (improving performance by 4.8% in median relative to the worst retrieval method), outweighing the importance of the ML model used (2.2% difference in median). Results show that relying on Jaccard Containment (i.e., the fraction of entities of the table to augment found in a join candidate) as a criterion is an effective way of retrieving join candidates that improve downstream prediction performance. The way join candidates are used to augment the base table has a noticeable effect on the prediction performance, although the larger difference comes from joining multiple candidate tables at the same time rather than only selecting the best candidate (a difference of 3.1%). Also, simpler aggregation methods outpace their complex counterparts in terms of speed (being up to 5 times faster) and experimental results indicate that the larger cost of more intricate methods does not yield proportionate gains in prediction performance.

To validate the generality of our findings, we contrast the insights obtained from the analysis carried out via YADL with those obtained from Open Data US, a well-referenced real data lake in academia [15, 33] over a range of input base tables. This comparative approach ensures that YADL upholds the complexity of real scenarios and is representative enough to be a benchmark.

After outlining the problem setting, enumerating the challenges and discussing previous work in Section 2, we share our main contributions:

(1) We develop YADL: a novel benchmarking data lake based on YAGO that allows to test retrieval and augmentation techniques in a controlled environment. YADL, the base tables, and the pipeline are available and easily extendable to spur further research. YADL is covered in Section 3.
(2) We implement a full prototype pipeline to model the problem of augmenting tables from data lakes. We track various metrics across the pipeline to measure the prediction performance, execution time, and RAM usage of the different methods. This is discussed in Section 4.
(3) In Section 5, we conduct an extensive experimental campaign testing 8 base tables over various data lakes, retrieval techniques, join selection methods, aggregation solutions, and ML models. Through the analysis of the results, we provide insights on the characteristics and impact of all the tasks involved in learning from data lakes.

We conclude the paper with a discussion of future research directions in Section 6.

2 AN END-TO-END ANALYTIC PIPELINE

Problem statement. Consider a user training a ML model to predict some quantity. The corresponding quantity appears in the training data as a target column Y of a *base table* T. Assume our user has also access to a large collection of tables (a *data lake*) $D = \{T_1, T_2, ..., T_m\}$, some of which may contain additional information that can enrich the base table T. Each of these tables T_k is a bi-dimensional collection of data organized in columns $C_k^i \in T_k$ that may include both categorical (names, codes etc.) and numerical data (price, revenue, tax rates etc.). While cross-table metadata such as foreign keys is not available in this setting, joining T with some of the tables in D would be beneficial for the target prediction task.

Given T and D, a table $T_k \in D$ is considered to be *joinable*, or a *join candidate*, if at least one of the columns $C_k^i \in T_k$ has a non-empty intersection with one of the columns in table T: given column $Q \in T$, column $C_k^i \in T_k$, $\exists\, C_k^i \in T_k \mid Q \cap C_k^i \neq \emptyset$.

The user is interested in optimizing the performance of the ML model on a collection of columns (or "features") X according to the quality of the prediction on the output space for the target Y (*e.g.,* the movie revenue). Columns in X may come either from T, or from joined tables in D.

In such a scenario, the user is likely to go over the following main steps: **Join Candidate Retrieval**, **Join Candidate Merging** (which includes the **Join Selection** and **Aggregation** tasks), and **Prediction**. Depending on the specific scenario, some of the tasks may be executed in a different order or not at all. In the following, we will drop Join from the names when clear from the context.

Finding the join candidates. Given a base table T and a data lake D, the **Candidate Retrieval** task consists in discovering join candidates ("Retrieval" in Fig. 1). This task looks for tables that can be considered as "candidate joins" for the given base table. Several systems tackle this problem [3, 9, 12, 26, 42, 44]. While integrating data is not a central focus of the ML literature, augmenting features via joins is recognized as key to ML [27, 35].

Though implementations may vary, retrieval methods share some similarities: 1) they start by indexing the data lake, and 2) they rely on a concept of "column similarity" along with a related metric. Generally, this metric is Jaccard Containment; however, other measures of set similarity may be used [42]. **Jaccard Containment** (JC) is defined as

$$\text{Jaccard Containment} \qquad \frac{|Q \cap C_k^i|}{|Q|}, \tag{1}$$

where $Q \in T$ is a query column, $C_k^i \in D$ is a candidate column in table T_k, $|Q \cap C_k^i|$ is the cardinality of the intersection between the two sets and $|Q|$ is the cardinality of the query set itself. Intuitively, if

this ratio is high, then a large fraction of Q is found in C_k^i, suggesting that the two columns should be joined.

Retrieval methods are typically designed to be used on large data lakes (potentially, up to millions of tables) and be as exhaustive as possible to maximize recall. However, three issues arise. First, these methods do not assess the relevance of join candidates to the query or the downstream task. While a large containment value may indicate that a join *can* be done, it provides no guarantee that this join is actually meaningful or useful. Second, the number of candidate joins could become too large for practical use. Manually identifying the best candidates may be excessively time-consuming, and performing all joins might be too expensive in terms of time/memory constraints. A user-defined threshold on the containment can be used to filter the most promising joins, but deciding the correct threshold is problematic. Third, Jaccard Containment does not take into account the cardinality of a column: in an extreme case, if a column $Q \in T$ contains only a single value, it would have perfect overlap with another column $C_k^j \in D$ that contains the same value. While this behavior may be desirable for some specific Information Retrieval tasks, it further expands the number of candidates with high containment. These limitations have led to the development of methods tackling the *Join Selection* task.

Merging the candidates. After the set of join candidates has been built by the retrieval task, the candidates need to be merged with the base table to augment it with their columns (**Merge** in Fig. 1). Although smaller than the data lake, this candidate set may remain too large to be handled directly due to resource constraints. This problem suggests the need for an additional filtering, namely the **Join Selection** task, with the goal of *identifying a subset of candidate joins that maximizes the prediction performance over a downstream task*. This task may be performed by removing joins that are not likely to be useful through the application of rules (and without executing the joins themselves) to the retrieval result [27, 38], or by testing each candidate joins to find those that bring benefit [4, 10, 15, 28]. In this work, we focus on the latter strategy.

It becomes therefore necessary to address the difficulties that come from joining base table and candidates. We consider ML models that involve a specific set of samples (rows) with their features. Augmentation operations in this scenario enrich said features while keeping the original set of samples constants, thus requiring a *left join*. A complication with left joins is that, for any un-joined row in the left table, null values will be added in the new columns. Alternative methods based on Full Disjunction [16, 25] would add irrelevant samples that should then be dropped.

It is not always possible to limit joins to one-to-one relations; often, we need to join on one-to-many or many-to-many relations. For example, to join a table about movies with one that contains ratings for the movies on the column "movie title", every movie with more than one rating would be involved in a one-to-many relation, and the content of all the rows with such relations gets duplicated. This standard behavior is problematic when the downstream task involves a ML or statistical analysis method, as such models assume that each row correspond to one sample, and the sample's features are found on that row. Thus, the **Aggregation** task bridges the result of the join with the downstream methods, combining the information contained from a potentially large number of rows into one. More precisely, given a tuple t in the base table T that needs to be joined with n tuples from a binary table (A, B) over A, the goal is to augment t with attribute B by selecting one value that represents the information coming from the n joining tuples. How to aggregate or select a representative value from the new attribute reminds traditional data integration problems such as truth discovery [8] or data fusion [1]. In this spirit, Deep Feature Synthesis (DFS) [23] takes a set of tables and a join plan to recursively create new features; it does so by aggregating the replicated instances using different functions (*e.g.,* average, median, mode).

Subject	Relation	Object
Paris	isLocatedIn	France
Paris	isLocatedIn	Europe
Rome	isLocatedIn	Italy
Rome	hasDensity	2236 / km^2
Rome	hasType	City

Table 1: Examples of YAGO triplets.

Learning on the augmented data. Finally, the integrated table is used to train a model to **predict** a target variable. This task can be deployed with easily available methods such linear regression/classification [36], or more complex methods such as CatBoost [37]. Though we present this task as distinct from the prior ones, an optimized pipeline may implement learning at intermediate sections of the pipeline to ensure the selection of the most appropriate candidate tables, contributing to improved model performance [15].

3 BUILDING YET ANOTHER DATA LAKE

In this section we detail the construction of YADL (Yet Another Data Lake), a synthetic data lake built starting from the YAGO knowledge base (KB). Our goal is to have a scalable and reliable data lake that allows users to evaluate the main steps of the pipeline in Fig. 1, while avoiding some of the confounding factors that come from working with unvetted data, such as typos, inconsistent schemas and data format, and other sources of noise. These additional challenges can easily be added to YADL by generating typos in entries of tables or randomly replacing column names by synonyms. Also, YAGO is a general content KB and therefore YADL can used to augment base tables belonging to different domains, as we report in Sec. 5.

3.1 YAGO: the source of the original data

YAGO [40] is a knowledge base (KB), i.e., a semantic database that contains knowledge about the real world. The KB is composed of triplets, or "facts". Triplets in YAGO follow the RDF standard, so that each triplet has a *subject* connected to an *object* through a *predicate* (or relation). Subjects and objects are considered to be *entities*. For example, in the triplet "Paris, isLocatedIn, France", "Paris" is the subject entity, "France" is the object entity, and "isLocatedIn" is the predicate (or relation) that connects them. The object need not be another entity: it may also be a lexical value such as the "population density". Each entity belongs to a set of "classes" (or "types"), arranged in a taxonomy; "Paris" belongs to the class "City",

YAGO City				
Subject	isLocatedIn_1	isLocatedIn_2	owns_1	popDensity
Paris	France	Europe	Tour Eiffel	null
Rome	Italy	null	Olympic Velodrome	2236

Table 2: Reshaped table produced by selecting entities with class "City".

subclass of "Populated place", itself subclass of "Geographical locations". Table 1 shows example triplets from YAGO. We use YAGO 3.0.3 [29], which includes facts up to 2022.

3.2 From knowledge base to relational tables

Information in data lakes is typically stored as tuples in tables, rather than the triplet format used in YAGO. This format is different from the typical tabular format found in data lakes: for this reason, we construct YADL by reshaping the triplets into binary tables. Wide-form relational tables can then be constructed by joining binary tables on the column that contains the triplet subjects. We assemble two YADL variants (Binary and Wordnet), which differ in the number of tables and in their shape (i.e., the number of rows and columns of each table).

Binary tables. In the binary variant, a new table is generated for every predicate in YAGO, such as "hasCapital" or "hasDirected". YAGO contains a limited number of predicates (70), so the result is a small data lake where some of the tables have millions of rows (e.g., "isLocatedIn", or "isCitizenOf"), and some have very few (e.g., "hasTLD"). Each relation table contains an attribute named "subject" and another attribute named as the actual predicate. For example, the triplet "France - hasCapital - Paris" leads to a table "hasCapital" with columns "subject" and "hasCapital"; finally, a row "France, Paris" is added to the table. This process is applied to all 70 relations and their triplets.

Wordnet tables. Tables in data lakes typically have more than two columns. To reflect this, we develop a second variant of the YADL lake where tables have a larger number of columns. In this case, we leverage the classes to which entities belong. We use the Wordnet [30] classes (e.g., "Person", "Company", "Artist"), as they are more generic than WikiData classes, whose higher level of granularity (e.g., "Treaties entered into force in 1994") is not necessary for our purpose. We select this particular subset of tables because it provides a relatively clean set of triplets to use, without excessive duplication. In principle, it is possible to create a larger number of tables by employing all classes, or select a subset of the classes to construct a subject-specific version of YADL. In total, we create 1015 Wordnet *seed tables*.

For each class, a new table is generated following the example in Table 2. Initially, the seed table includes solely the "Subject" column (e.g., the "City" table includes values like "Paris," "Rome," etc.). Subsequently, we join every relation associated with subjects in the table: in Table 2, the subjects with type "City" are joined with relations "isLocatedIn", "owns", "hasPopDensity".

This transformation of triplet tables into wide format tables leads to tables with null values because not all entities of a given type have the same relations. For example, column "hasPopDensity" contains null values for subject "Paris" in Table 2.

Addressing one-to-many relations. An issue that arises from working with data stored as triplets is handling subjects that are linked to many objects through the same predicate: we give an example of this in Tab. 1, where "Paris" is linked to two objects via the predicate "isLocatedIn". We choose to address it in two different ways in the two versions of YADL: in the case of Binary, triplets that share the same subject and predicate are transformed into tuples that share the same value in the "subject" column, and have different value in the predicate column (e.g., given Tab. 1, the resulting binary table will have columns "Subject" and "isLocatedIn" and it will contain tuples (Paris, France) and (Paris, Europe)); for the "Wordnet" variant, we flatten the group of objects by creating new columns, thus moving them on the same tuple rather than splitting them (e.g., in Tab 2, column "isLocatedIn" is flattened over "isLocatedIn_1" and "isLocatedIn_2"). While this problem is an artefact of the creation process, it is analogous to the aggregation task already described.

As a result of handling these relations differently in each version of YADL, the aggregation step in the pipeline affects them in different ways, thus exposing different problems.

Sub-tables. We create supplementary tables derived from the Wordnet tables to augment the table count in the data lake. In real-life lakes, numerous versions or variants of the same table are common, encompassing collections of tables that undergo slight modifications over time [19]. This redundancy poses a challenge for information retrieval methods in distinguishing between pertinent and extraneous tables. Hence, it constitutes a crucial aspect in a benchmark data lake like YADL.

We systematically generate sub-tables for each "class table" by considering all possible combinations of arity 2 and 3. For Tab. 2, these combinations would include "(isLocatedIn_1, owns_1)", "(owns_1, popDensity)", and "(isLocatedIn_1, owns_1, popDensity)". Then, we evaluate each combination by selecting only the columns involved in it. To maintain a minimum size threshold, we exclude all combinations with fewer than 100 rows. Rows containing null values are retained, provided that the column combination involves at least one non-null value.

Sub-tables may include unrelated columns, or columns that are not relevant to the downstream task. Another side effect is that many of the generated tables are similar to each other, as they share the "subject" column at the very least, and possibly columns in the combination. This matches real data lakes where many tables are not related or useful for the downstream task, and yet there may be a significant amount of redundancy (*e.g.,* because of partly redundant extraction of primary stores).

Statistics on the two variants Binary and Wordnet are reported in Table 3. The data lakes are available at https://zenodo.org/doi/10.5281/zenodo.10600047, while the code to prepare YADL is at https://github.com/rcap107/YADL.

4 IMPLEMENTATION OF THE PIPELINE

We now discuss in detail the different steps of the pipeline introduced in Figure 1. For the "Retrieve" step, we describe the advantages and disadvantages of each candidate retrieval method; for the

"Merge" step we propose different join selectors and aggregation methods, and we explain how the two parts of this step are intertwined; finally, for the "Predict" step we go over the ML methods used to test the prediction performance.

4.1 Retrieving the candidates

The first step of the pipeline retrieves tables that can likely be joined into the target table. We consider retrieval methods that work by taking a query column and return a – possibly ranked – list of candidates. Retrieval methods involve offline preparation steps that build data structures summarizing the data lake, and a querying step where the candidate joins are retrieved. Preparation steps may require the construction of sketches [13, 43], indices [42], or the exhaustive measurement of Jaccard containment for every column in the data lake; the preparation step also involves persisting the data structures on disk for future use.

We focus on retrieval strategies estimating Jaccard containment (Equation 1): a candidate column is considered as a good join if it contains a large fraction of the entities of the target table (Section 5.2 provides an empirical justification). We consider two different complementary strategies, an exact and a stochastic approximation, as well as a "hybrid" approach combining them. The rest of the pipeline is transparent to the retrieval method: given a list of join candidates, it will test each candidate regardless of how the list was constructed. Depending on the chosen method, the number of candidate joins may still be too large (potentially, in the thousands) to be handled in the merge step, either due to limitations on the execution time or memory: for this reason, a likely scenario would be selecting only those candidates whose Jaccard containment is larger than a certain threshold, or the top-K candidates proposed by the retrieval method. For our experiments, we reduce the number of candidates to the top-30, as we observe that almost all candidates with high containment are within this limit (Fig. 7).

Exact Matching. We compute *Exact Matching* (Exact) by exhaustively measuring the exact Jaccard containment between each query column and every other column in the data lake. This can be implemented efficiently by first building a "vocabulary" on the target column, and then scanning the whole data lake to compute containment. Effectively, this builds a "gold standard" for any method that relies on Jaccard containment as it will provide the candidates that have the exact best containment across the entire data lake. Having an exact value for the containment allows to easily rank candidates according to that, thus providing a better way of selecting either "top-K" best candidates in the pool, or to apply a threshold and select candidates based on that.

Naturally, this method has a major drawback: it requires the computation of the exact containment value for each pair (query column, candidate column) in the data lake (with a cost that depends directly on the size of the data lake and the tables therein), and this operation must be repeated for every new query column. As we show in Fig. 10, the cost of repeating this operation for multiple query columns quickly adds up. A more fundamental question is that the Jaccard containment is probably a proxy of the actual utility of joining on a table. Thus computing its exact value may be a waste of resources. We explore the value of containment in more detail in Sec. 6.

MinHashLSHEnsemble. MinHashLSHEnsemble [44] (MinHash) requires an indexing step before executing the queries and relies on Locality Sensitive Hashing (LSH) to find tables whose Jaccard containment is larger than a threshold set by the user when the index is initialized. At query time, given a query column MinHash returns all candidate columns whose containment is approximately larger than the threshold. MinHash is remarkably robust to noise in the input data, such as typos, mismatch in surface forms and inconsistencies in the schemas of the tables to index. Its recall is also quite high. Once the indexing step is done, the querying operation itself is very fast.

MinHash has some drawbacks of its own: 1) As the only information that MinHash can provide is that a candidate has a containment larger than the threshold, it is impossible to rank candidates within this pool, or to select them based on a threshold other than that used to build the index. 2) Since MinHash returns an approximate result, it may happen that candidates that have a measured containment lower than the threshold are proposed: indeed, when the MinHash query results are compared with the results obtained from *Exact*, MinHash produces a large fraction of False Positives. 3) Finally, the index cannot be updated: any change in the data lake requires re-indexing, which can take a long time.

Hybrid MinHash. We discuss a third method that combines Exact and MinHash, to provide insight into how to mitigate some of their issues and improve downstream performance. *Hybrid MinHash* employs MinHash to find candidates for a given (possibly unknown) query column, then uses Exact to re-rank all candidates retrieved by MinHash. This method leverages the stateless nature of MinHash by producing results for any query, without requiring any prior operation; it then refines the results by measuring the exact containment to produce a better ranking of the candidates. As the size of the candidate pool is much smaller than the entire size of the data lake, measuring the containment is much faster than indexing the entire data lake using Exact. Like Exact, it is also possible to rank the candidates and use a threshold. As we show in the experimental section, the prediction performance of Hybrid MinHash is quite similar to that of Exact Matching.

Just as it takes some of the strong suits of both MinHash and Exact, Hybrid MinHash also retains some of their disadvantages: 1) just like MinHash, any evolution in the data lake require rebuilding the entire index. 2) As the first filtering relies on the MinHash results, False Positives add needless cost and any candidate missed by MinHash is lost. 3) Re-ranking candidates is done when querying by measuring the exact containment, which increases the query time substantially.

Implementing the retrieval methods. The performance and resource utilization of retrieval methods is largely dependent on their implementation. We rely on the Python implementation of MinHash provided by the Datasketch package [43], while we implement Exact matching in Python using the Polars package [41]. Data structures are stored by persisting on disk the *MinHashLSHEnsemble* for MinHash and the candidate ranking for each query column for Exact; Hybrid MinHash relies on the MinHash data structures to work, so no additional storage is required for it. Table 3 reports the size on disk of the data structures used for either method. The Datasketch implementation of MinHash does not provide an online

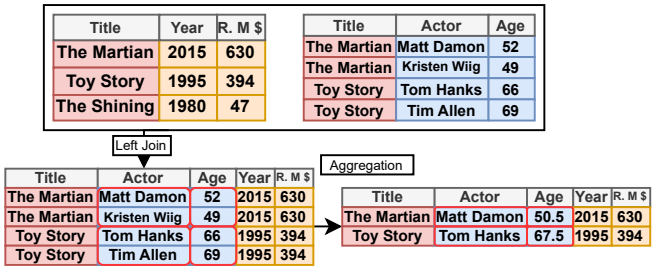

Figure 2: Example of how a left join would duplicate rows from the base table.

interface to query with, which means that the index must be loaded before querying: this adds a substantial overhead to the "actual" query time, as reported in Fig. 6.

4.2 Merging base table and join candidates

In this step, the candidates that were retrieved in the previous step are combined into a new, *integrated* table that joins the information in the base table T with additional information sourced from augmentation tables. As we highlight in Fig. 1, this step combines two operations that cannot be executed independently: **join selection**, and **aggregation**. We are considering a scenario where we train a machine learning model using a specific input table, which means we have a fixed set of training samples. Consequently, we must use left joins to ensure the merged table maintains a constant number of rows. Additionally, we need to aggregate one-to-many relationships to prevent the sample replication that would occur with a typical join.

The join step is a critical part of the pipeline, as joining two tables is already an expensive procedure. Indeed, joining all candidates at once may lead to issues with time and memory; moreover, since not all joins are always beneficial [27], we are interested in joining only those that bring actual benefit in the downstream task.

For these reasons, optimizing this step seems important, and one way of doing this is selecting which candidates provide the most benefit when they are used to augment the base table. Selecting these candidates may involve performing additional (intermediary) joins prior to executing the full join, bringing join execution and aggregation into play once again.

Aggregation. As mentioned above, when joining tables, one-to-many relationships must be aggregated to avoid replicating samples in the base table which would modify the initial data sampling.

Following the example in Fig. 2, a join on "Title" leads to the duplication of the two first rows in the base table, as both title values appear in two rows of the candidate table. Our objective is the prediction of the "Revenue" values for a given movie, characterized by its title. Which of these rows should be used for this prediction?

We test three different aggregation strategies in "**first**", "**mean**", and **DFS**. With "first", any row where the values belonging to the base table are duplicated is dropped from the joined table, with the exception of the first copy. With "mean", each categorical attribute is replaced by the most frequent value (the mode), and each numerical attribute is replaced by the mean of all values in that attribute. DFS is functionally a more powerful and exhaustive "mean", as it uses additional aggregations (median, standard deviation. etc.) rather than only the mean and add new features for each aggregation.

Join selection. We implement and test four different *join selectors*. All selectors follow the fit-predict paradigm proposed by scikit-learn [36]: during the fit operation, the training split of the base table is passed to the selector, which is used to train an internal state which comprises the list of candidates that should be joined, as well as a model trained on the training split of the data. Internally, joins are tested on a validation split that is sampled from the training split to avoid overfitting. In the predict operation, the test split of the base table is passed to the selector, which then outputs a prediction based on the test split joined with the selected candidates. Naturally, the selection of the candidates depends on the specific selector that is being used. An advantage of following this paradigm is that it makes extending the pipeline with new selectors very simple.

The reference selector is **No Join**, where no join candidates are provided and the ML model is trained directly on the base table. **Highest Containment Join** measures the exact containment of the candidates provided in the retrieval step, then re-ranks them by that. The candidate with the highest containment is selected; ties are broken by taking one candidate at random. This selector makes no assumptions on whether the candidates are ranked in the retrieval step or not. **Best Single Join** iterates over each candidate one at a time, then trains a separate ML model for each joined table, evaluating the performance with a validation split. The single candidate that provides the largest benefit on the validation split is then joined with the entire training table to re-train the ML model; during prediction, it is joined on the test split and used to predict the result. **Full Join** blindly joins all candidates provided during the fit, then trains the ML model on the resulting table; the same join operation is repeated in the predict step on the test split of the base table. Finally, **Stepwise Greedy Join** keeps more information than what is possible using only one table, while avoiding the risk of joining tables that are irrelevant to the task which comes with Full Join. This selector re-ranks all candidates like in the Highest Containment case, then iterates over every re-ranked candidate. In each iteration, the candidate is joined on the current table, then its performance is evaluated. If the newly joined table improves over the previous, it is kept as is and becomes the current table; if the performance worsens, the candidate is discarded.

Highest Containment and Best Single Join produce smaller output tables as they only join *one* candidate, rather than *all* potential candidates like Full Join and Stepwise Greedy Join: we can therefore classify the two pairs as **single-table selectors** and **multi-table selectors** respectively.

In all selectors that involve a join, aggregation is carried out prior to executing the join itself. [1]

4.3 Predicting with a ML model

The learning step is implemented by inputing the final table to a supervised-learning regressor. We evaluate two different methods:

[1] In our pipeline, aggregation is carried out before executing any join by grouping the "right table" by the join key, then applying one of the aggregation functions described above. This is to avoid materializing large joins, which have a huge cost in memory and time. Due to how aggregation is carried out, the final result is the same before and after the join, so executing it before materializing the joined table is more efficient.

Statistic	Wordnet	Binary	Open Data US
Data lake size	9648	301.0	3886
Minhash index size	441	1.4	415
Exact matching index size	11.3	0.03	5.43
# tables	32103	70	5591
Avg. # rows	287134.33	17124.50	22343.92
Avg. # columns	2.00	23.86	3.17
Avg. # categorical columns	1.70	12.76	2.78
Avg. # numerical columns	0.30	11.10	0.39
Avg. frac. nulls	0.00	0.09	0.31

Table 3: Statistics for the data lakes, along with the size (in MB) on disk of the data structures of the retrieval methods.

Base Table	# Num. Att.	# Cat. Att.	# Rows
Company Employees	2	7	3109
Housing Prices	3	7	22250
Movie Ratings	7	10	3837
Movie Revenue	8	10	3837
Schools	4	3	1774
US Accidents	3	9	5222
US County Population	1	1	3059
2020 US Presidential Results	3	6	22093

Table 4: Statistics for the base tables.

a simple linear regressor/classifier and CatBoost [37]. Our intent is comparing the performance and resource expenditure (in time and memory) of either method: linear models are very simple and inexpensive compared to a gradient boosting method like CatBoost. We choose CatBoost as our "complex and effective" method because gradient boosting methods are effective at working with tables [18], and CatBoost in particular is good with categorical data, which is common in our scenario.

4.4 Evaluation

We evaluate the full pipeline. The efficacy of these steps can be measured in the context of modern data analysis tasks, which generally fall into either predictive (training a machine learning model) or prescriptive (causal inference to answer what-if and how-to questions) categories [15]. Because it can easily be evaluated quantitatively, we focus on measuring the performance of a predictive model trained on the base table alone or on integrated tables. Additionally, we track memory and time requirements in various steps of the pipeline.

The code for the pipeline is available at https://github.com/rcap107/benchmark-join-suggestions.

5 EXPERIMENTAL STUDY

For our experimental campaign, we test the different sections of the pipeline over three main dimensions: prediction performance (either R^2 score for regression, or AUC for classification), execution time, and memory consumption. Focusing on multiple dimensions allows us to have a view of the different trade-offs across methods.

Data lakes. We use the two YADL lakes (as detailed in Section 3) and Open Data US, a data lake widely employed in the literature [15, 42, 44]. Their statistics are reported in Table 3.

Base tables. We evaluate six tables from sources that are not related to the lakes: Company Employees (predict "number of employees"), Movie Revenue ("revenue of movies"), Movie Ratings ("ratings of movies"), US Elections ("fraction of votes by party in each US county during the 2020 elections"), US Accidents ("number of accidents by US county in the year 2019"), Housing Prices ("price of a house").

We also include one "internal" table derived for each lake: US County Population (from YADL, predict "population of each US county"), Schools (from Open Data US, "school classification"). For all datasets, the query column values must be matched with the entities in YADL using semantic annotation solutions [21, 34]. In our experiments, we have done the match manually to remove the noise from this task. Also, query columns are chosen based on what a user may reasonably consider as "key" (e.g., the movie title). We include internal tables as those have been used in several previous works for evaluation [4, 15]. Datasets' statistics are reported in Table 4.

Default parameters. All experimental runs are executed over 10 cross-validation splits: the test fraction is 0.2 both in the outer cross-validation split and in any join selector that requires a validation step for its fitting step. Experiments were executed on a cluster with 72 CPUs and 376GB of RAM. The implementation of the pipeline is in Python; aggregation and join operations rely on Polars [41] as backend, while the ML models are implemented using Scikit-learn [36] and CatBoost [37]. To reflect the experience of a data scientist that needs to construct a meaningful table starting from a data lake, and to highlight the effect of joins on the downstream task, we run experiments on a depleted version of the tables, i.e., the input tables include only the primary key column and the target column.

Retrieval We use a containment threshold of 0.2 for the preparation of the MinHash index, and clamp the number of candidates returned by each retrieval method (Exact, MinHash, and Hybrid MinHash) to 30. These values were chosen to balance execution time and expected number of candidates given the distribution of containment encountered in the different data lakes (Fig. 7).

Selection We fix the number of iterations of Stepwise Greedy Join to 30: this number is consistent with the number of candidates that are provided in the retrieval step. None of the other join selectors have features parameters to tweak.

Learning We fix the number of CatBoost iterations to 300; we stop training the model 10 iterations after the optimal metric has been detected; we set the L2 regularization coefficient to 0.01.

5.1 Retrieval is the most impactful pipeline step

Our first goal is to pinpoint which steps of the pipeline have a significant impact on the studied dimensions (if there are any): optimizing these influential sections can yield the greater benefits.

For each pipeline task (**retrieval**, **selection**, **aggregation**, **learning**), this process involves separating the variance that is due to that specific task from that due to the dataset *and* the other tasks. This is achieved by aggregating variables other than the one of interest.

For instance, when analyzing the impact of the **retrieval** step, results are grouped by **join selection model**, **aggregation method**, and **ML model** before averaging prediction performance and run time. Then, the results obtained with each value in the variable of interest (e.g., Exact, MinHash and Hybrid MinHash) are plotted as a difference from the average method for that variable (the "average retrieval method"). When dealing with binary variables, such as the ML model, the "best" method is identified using the R^2 score and compared to alternative. We report the difference in R^2 and relative execution time with respect to the average method for retrieval (Fig. 3a), selection (Fig. 3b), aggregation (Fig. 3c), and learning (Fig. 3d). Individual runs are reported as dots; runs are assigned distinct color palettes depending on their data lake.

Retrieval. We observe that Exact Matching shows the best overall prediction performance (2.55% gain in median across all experiments), while MinHash is showing substantially worse results (-2.26%) and Hybrid MinHash sits in the middle between the other two methods. MinHash is also faster than the other two, despite the fact that the number of candidate joins is the same for all methods. This is due to the fact that, on average, the candidates retrieved by MinHash have a much lower containment than those proposed by the other two methods (Fig. 5a): this results in a shorter join and train time due to the smaller amount of data that must be moved and used for fitting the models. The difference between Hybrid MinHash and Exact Matching is likely due to the fact that MinHash does not have perfect recall: some of the higher containment joins are replaced by candidates with lower containment, which reduce the overall training time and the prediction performance. Overall, the two methods based on precise ranking (Exact and Hybrid) outperform the method based purely approximate matching (MinHash), suggesting that the larger computational cost is worth it.

Selection. The choice of selector shows a noticeable effect when moving from single-table selectors (Highest Containment and Best Single Join) to multi-table selectors (Stepwise Greedy and Full Join), bringing a benefit close to 3% in median (Fig. 3b).

Stepwise Greedy Join is an outlier in how slower it is compared to all other methods. This is not surprising, since this selector executes a join and trains a model in each iteration, then re-trains the model at the end of the fit step. This selector is expected to yield better results than Full Join by avoiding candidate joins that do not bring benefit. Fig. 3b shows that this objective is not achieved, which makes the much longer execution time even more concerning.

Highest Containment is significantly faster than all other solutions because it only re-ranks candidates before joining the best. Best Single Join trains and evaluates a model for each candidate, before selecting the former.

The slight difference in performance between two single-table selectors is consistent with the expectation that the optimal join is likely to exhibit high containment, and will thus be selected by both selectors. However, results show that the candidate join with the highest containment is not necessarily the best overall, suggesting that relying exclusively on Jaccard Containment is not sufficient to maximize the prediction performance. On the other hand, the more significant difference between single-table selectors and multi-table selectors is explained by the learning model benefiting from an increased set of features: merging more than one table inherently results in a richer feature set compared to the scenario where only a single table is joined.

An important observation is that all selectors rely on the candidates proposed by a retrieval method: if these candidates have poor quality, the selectors cannot compensate for that.

Aggregation. Among the pipeline steps under evaluation, aggregation exhibits the least variance in prediction performance between methods. However, the situation changes when considering the execution time: more complex aggregation methods are several times slower than the much simpler "first".

DFS does show slightly better results on average compared to the other methods, owing to the creation of new features; however, it is extremely slow compared to the other methods. It also creates new features greedily, which may lead to memory issues.

Experiments reported in Fig. 3c were obtained using only a subset of the datasets, and include only the single table selectors because DFS ran into scalability issues with Full Join and Stepwise Greedy. Combining DFS with the Stepwise Greedy selector would skew the time difference plot even further.

Learning. The comparison between learning methods yields predictable results: CatBoost outperforms the linear model (2.29% in median); it is however close to two times slower in median, likely due to the fact that some runs do not converge and early stopping does not help. This outcome is not very surprising, since CatBoost is a complex, state-of-the-art model. Additionally, CatBoost has a significantly larger memory footprint compared to the linear model (Fig. 11). Due to the clear trade-offs in this step, the user should make their model choice based on these considerations and their requirements.

Different Lakes. Figure 4 reports the aggregate results obtained on the different data lakes. In this plot, we report the *absolute gain* from the un-joined base tables (rather than the relative difference from the average method like in Figure 3), to highlight the potential gains in prediction performance that can be achieved by augmenting them with candidate joins. The figure also shows the risk involved in joining sub-optimal candidates (shown by the negative prediction performance). Open Data US shows both the best and worst prediction results overall. Poor results can be explained by the inherent "dirtiness" of the data lake, especially when compared to YADL: this results in a lower average containment compared to YADL (Figures 5a and 7). On the other hand, the good results can be attributed to the "Schools" internal dataset, which achieved perfect classification performance in most cases and skewed the results in that direction; this is also visible in Figure 5b, where the results with the highest containment are also those with the best performance. YADL's internal table (US County Population) exhibits a similar behavior to that of Schools. This is explained by the fact hat internal tables, i.e., tables that are sampled from the data lake itself, tend to have copies in the data lake, which are likely to contain information that is correlated with the prediction task. This highlights the specific behavior of internal tables, which are routinely used in the experimental campaigns in the literature.

An advantage of YADL is that it is built based on a KB that contains "general knowledge" notions: as a result, external tables

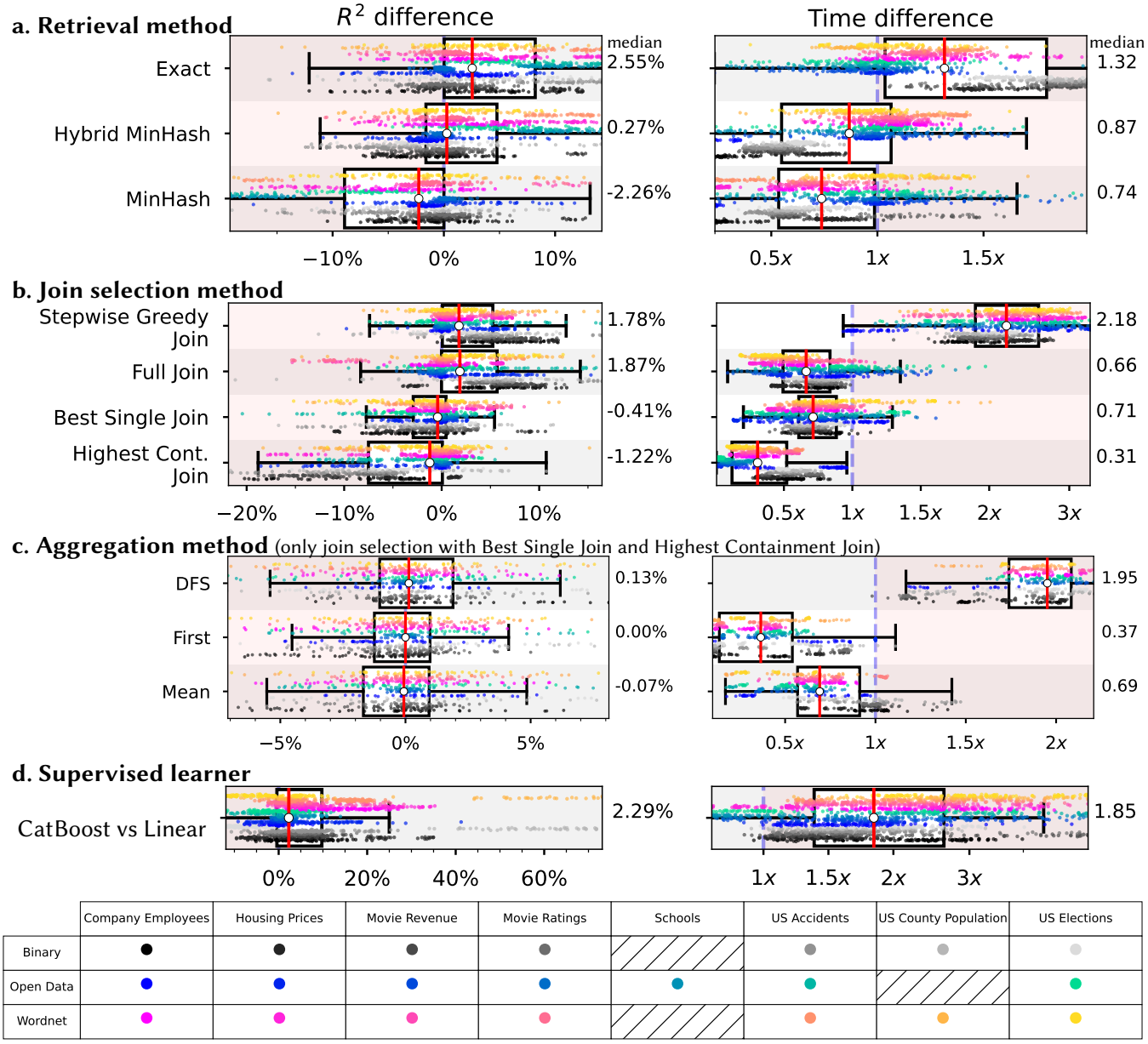

Figure 3: Experimental results across all data lakes, comparing the performance difference between evaluated methods in different steps of the pipeline. The median difference is reported on the right of the plot. (a) reports the difference between retrieval methods, (b) compares join selectors, (c) compares aggregation methods and (d) compares ML models. In all cases, results are reported in relation to the "average method".

are more likely to find matches in YADL than in Open Data US, whose tables contain mostly US-based demographic data.

5.2 Containment affects the entire process

We now focus our attention on one of the steps with the larger impact on both prediction performance and execution time: retrieval. Indeed, Fig. 3a shows how re-ranking with exact matching outperforms the approximate matching-base MinHash.

Fig. 5(a) shows the containment results for the tables obtained by the different retrieval methods. Exact largely outperforms MinHash, so we use it in the following analysis. It is also evident how, in general, retrieval on YADL Wordnet yields queries with a far larger average containment. We report top-200 containment to highlight how, even when retrieving a quite large number of candidates, the average containment of MinHash is very low when compared to the other two methods. Fig. 5(b) shows how the average prediction

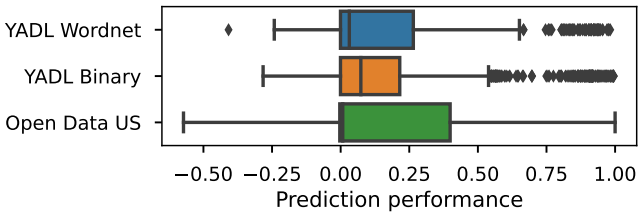

Figure 4: Comparison in prediction performance for the different data lakes. The performance metric is R^2 for regression and AUC for classification.

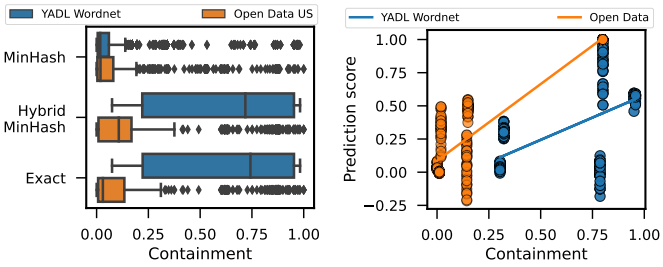

Figure 5: Relationship between containment and performance. (a) Average top-200 containment by query column with different retrieval methods. (b) Prediction performance wrt containment.

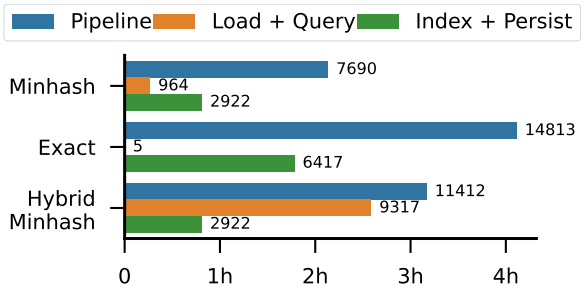

Figure 6: Comparison of time spent (in seconds) preparing a set of experiments with the different retrieval methods.

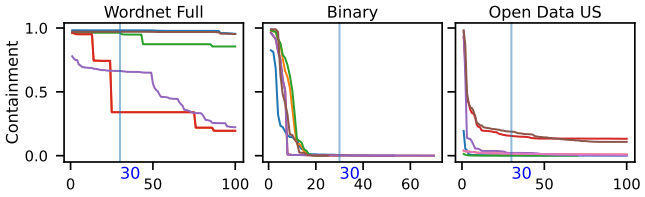

Figure 7: Exact containment measured over the top 100 join candidates for each base table on each data lake. The x-axis reports the rank, rank 30 is highlighted as it is the retrieval cutoff we use in our experiments.

performance follows closely the measured containment. In other words, higher prediction results occur more frequently when the containment is higher. When the join is exact (like in our scenario), having a high Jaccard Containment is very important to achieve good results. This is because left joins between tables that have very low overlap result in the addition of new features that have mostly empty values; such features are less useful than high containment joins, which augment a large fraction of the base table.

On the flip side, working with a larger set of values has the consequence of increasing the join and training time (Fig. 6).

Looking at the different retrieval methods, it is evident that the query results proposed by MinHash have a much lower average containment (Fig. 5a): this is due to the fact that MinHash does not provide a detailed ranking of the candidates, which may lead to losing track of candidate tables that have a high containment if the query result is sampled in some way. This is likely to be the case if the user has a certain budget of candidate tables to work with. In fact, MinHash has a threshold that can be tweaked to reduce the number of candidates that are retrieved, however we have observed that recall drops sharply at high thresholds.

While Hybrid MinHash appears to be faster than Exact Matching in the pipeline, it is important to note that re-ranking candidates incurs a non-negligible additional cost (Fig. 6).

5.3 Smart aggregation brings marginal benefits

While containment impacts heavily both prediction performance and execution time, aggregation has a similar impact on the execution time without the same degree of improvement for the prediction performance (Fig. 3c). Aggregating values always leads to a loss of information: in exchange for a larger cost, complex aggregation methods that preserve more information (DFS) or that replace values with better representatives of a group (mean) should lead to better prediction performance.

However, this is not what we observe: more intricate aggregation methods do not pay off their larger computational requirements. This is exacerbated by the fact that aggregation must be performed whenever a join is executed *at any point in the pipeline*, including joins executed during join selection. The result is a compounding slow-down of the entire pipeline, which increases the total execution time by many times compared to the basic method (first).

A possible explanation is that YADL data is categorical-heavy (Tab. 3), and features few numerical values. Categorical features are aggregated by using the mode, which retains less information than the mean. Working with candidate tables that feature mostly numerical values may lead to benefiting more from the "Mean" and "DFS" methods.

Poor performance of DFS are also explained by the fact that we are not fully exploiting its capabilities. We consider only join chains of depth 1: at each aggregation step, we join the base table with an additional table, rather than leveraging the recursive generation of features that is provided by DFS. As a result, DFS is not as effective at generating features as it would be with deeper join paths [5].

5.4 Tradeoff between analytic performance and compute cost

Execution time. Fig. 8 reports a breakdown of where the time is spent by different join estimation methods, while Figure 9 reports

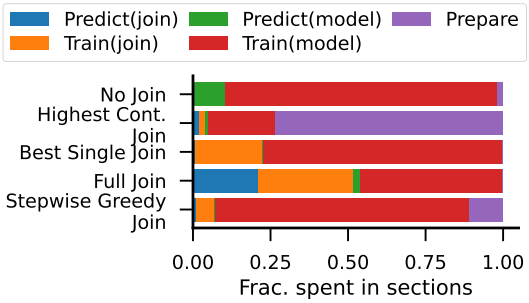

Figure 8: Breakdown of where time is spent for each selector.

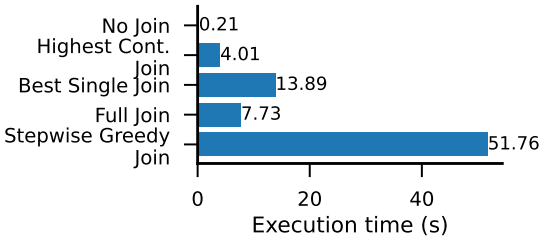

Figure 9: Average execution time for each selector.

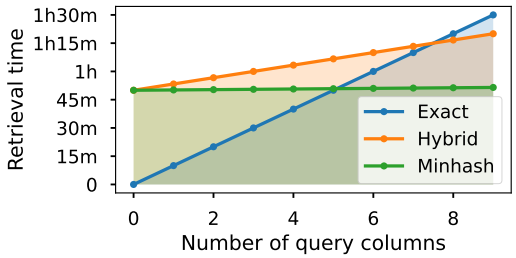

Figure 10: Evolution of the runtime as a function of the number of query columns for the different retrieval methods.

the total time required for a run on average by selector. *Prepare* tracks the time spent building data structures and loading data, including loading candidate tables in memory and building the containment ranking. *Train(model)* and *Predict(model)* track the time spent inside the ML model for training and prediction, respectively. Finally, *Train(join)* and *Predict(join)* track the time spent executing a merge operation, combining join and aggregation. Results are aggregated over all experiments, however aggregation is fixed to "first" because "mean" and "DFS" increase the join time to such an extent that other costs are almost negligible. The time distribution follows our expectations: most of the time is spent fitting models, with time spent joining (and aggregating) in second place. Highest Containment Join spends a relatively long time in the "Prepare" step due to the need to re-rank candidates before joining: since the join and train steps involve only one table, they are faster in comparison. Stepwise Greedy Join has a similar re-ranking step, however training the models in each iteration dominates the other steps. Full Join involves merging all candidates at the same time, then training a single model on the result: this explains how the fraction of time spent joining is comparatively larger than in other methods.

Figure 6 reports the time required to prepare and run the entire set of experiments. Tracking an entire experiment is needed to provide a "fair comparison" between the different retrieval methods and evaluate the time spent preparing their data structures ("Index + Persist"), as well as the time spent querying them to obtain the candidate joins ("Load + Query"), that are then used in the pipeline. Since the results of querying a column in a table can be reused across different experiments, the time reported as "Pipeline" is obtained by running a full set of experiments with default parameters and a specific retrieval method over all base tables. The total time is then compared with the time spent creating and querying the respective retrieval method.

This combined representation highlights the cost involved in computing the exact overlap for the given base tables: a single column requires about 600 seconds (on average) to be executed, and this cost increases linearly with the number of columns that must be queried. Conversely, building the MinHash index takes much longer than a single column, but it is a one-time operation to be executed on the lake and scales with the size of the data lake itself. Hybrid MinHash reuses the basic MinHash index and has therefore the same creation time; query time is much longer, however, owing to the need to re-rank the query results.

Overall, the time spent performing retrieval depends mostly on the number of performed query operations, rather than on the indexing operation. In our experiments, the time to measure the exact containment on the Wordnet data lake overtook the time for building the MinHash index after about 5 queries. This is also a consequence of how MinHash is designed specifically to minimize query time [43, 44].

The values used in Fig. 10 are obtained assuming that query retrieval time increases linearly with the size of a table and that the cost of creating the MinHash index is fixed for the data lake at hand. It shows how the fixed cost involved in building the MinHash index pays off after as few as six queries on average thanks to the fast query time. Hybrid MinHash requires more queries to break even due to its slower query time, yet it remains faster than Exact Matching when the number of query columns is larger. While the assumptions may not hold in general, the plot gives a reasonable estimate of the break-even points.

Although dependent on implementation, another factor that should be considered is the size on disk of the indices: the MinHash index occupies a much larger space on disk compared to the data required to hold the Exact matching ranking (Tab. 3).

Concluding, the choice of the retrieval method falls on the user. Exhaustive computation of the containment is a net gain in performance at the expense of an execution time that increases quickly as the number of columns to query increases. This may not be a problem if the user is aware of which columns should be queried; if, instead, the user is trying to conduct an exhaustive search over all columns, a method such as MinHash should be favored. This is consistent with the observations in [42]. Finally, in scenarios where the query table and the data lake do not change, query results can computed offline and reused; in these scenarios, the additional

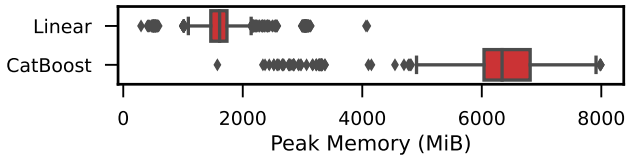

Figure 11: Peak memory usage by ML model.

cost of Exact Matching would be less problematic. In situations where the data lake tends to evolve over time, methods that support updating the index such as [3] or [12] should be considered.

Memory footprint. Memory profiling in Fig. 11 shows that peak memory usage is dominated by the ML model used in the prediction task. Difference across join selection methods is smaller compared to the difference between using a simple model (Linear) and a complex one (CatBoost). Another main factors in memory usage is the size of the candidate tables that must be joined.

5.5 Take-away messages

We summarize the main take-away messages from the experiments.

(1) Retrieving the correct join candidates is the most important task of the pipeline; good retrieval yields higher quality candidates, which improves performance in later steps (Figures 3a, 5a, 5b).
(2) Join Selection does not affect the downstream performance as much as the retrieval method (Figure 3b), and cannot compensate for poor retrieval performance. Joining all candidates from the retrieval step is the best trade-off between resource utilization and downstream performance.
(3) Complex aggregation methods are much slower than simpler ones and do not result in commensurate gains in prediction performance (Figure 3c). Furthermore, complex methods scale worse with multi-table selectors.
(4) Regardless of the ML model, execution time is dominated by the training when the aggregation is simple, while the inverse is true with complex aggregation functions (Figure 8).
(5) Hybrid retrieval methods led to prediction performance comparable to exact matching (Figure 3a) and scale better with many query columns (Figure 10).
(6) In general, joining candidates brings benefits over working with the base table (Fig. 4). The specific characteristics of a data lake (Table 3, Figure 7) affect many facets of the results (Figures 4, 5).

6 CONCLUSION

In this work, we build a synthetic data lake based on real data contained in a KB to use as test bed for evaluating methods to augment user-provided tables. We implement an easily-extendable augmentation pipeline to test the different steps of the augmentation procedure and the different algorithms for each of their tasks. We provide the benchmark data lake, the base tables, as well as the code required to extend it. Our results uncover a number of observations that we believe could help with directing research on this subject.

Indeed, various points remain unaddressed and can be the focus of extensions and future work.

More Lakes. Our experiments show that YADL Wordnet is representative of real data lakes. However, it mostly features categorical data. This characteristic partially explains the poor performance of Mean and DFS in the aggregation task. YADL can be extended by plugging new sources or by changing the functions used to generate sub-tables. Future work could include deploying more YADL variants, possibly with domain-specific data lakes [39].

More Tables. In our scenarios the tables are small enough for Full Join and Stepwise Greedy Join to be viable. Working with larger tables would stress the pipeline and force the use of methods to optimize resource use and achieve scalability. While we report more base tables than any previous work, it would be valuable to extend such a pool to cover new domains, according to the tables available in the lakes. An orthogonal problem for base tables and lake tables is their evaluation under the lenses of the ethical issues in their selection [31].

More Methods. In our work we do not try to be exhaustive over the possible methods as we believe the implemented ones are representatives of the recent literature for this problem. However, there are more design choices that have an impact on the performance. We report here a few examples of methods that could be deployed and tested in our pipeline.

The selectors in our pipeline *join a full table at each iteration*. This has the benefit of adding multiple features without executing the same join multiple times, but it has the risk of adding noise by introducing irrelevant columns. A possible solution is executing joins that add one column at a time [15]. Alternatively, methods such as [10] could be used to select retrieval candidates with high correlation.

It has been shown that data lake table *profiling* can bring in benefit in selecting joins and augmenting data [14, 15]. Profiling helps reducing the search space and directing retrieval methods towards meaningful candidates.

Similarity Joins. As in almost all work, we focus on the scenario where it is possible to perform joins with equality. However, this is not possible in general. Joining methods can be extended to overcome the issues that may arise in practice, such as typos, different formats, and different granularity. While similarity joins and semantic matching would help with this problem [9, 22], they would also add another dimension to the analysis of the results.

Chain of Joins. To limit the search space, we keep our chain of joins for augmentation limited to one. However, some of the methods have considered chains of join for augmentation [15]. Enabling join chains would also make clear the benefit of recursive methods such as DFS [5].

Augmentation Beyond Joins. Finally, another popular augmentation technique is "table unionability": tables are considered unionable if they have the same structure, i.e., they have the same number of columns with compatible data types [2, 24]. Also, papers tackling table discovery over data lakes includes other approaches that are not driven by the base table and joins [11]. For example, some methods are driven by user keywords [19] or by the embedding of the table [7, 20].

ACKNOWLEDGEMENTS

We acknowledge the invaluable insight and expertise provided by Léo Dreyfus-Schmidt and Du Phan (formerly from Dataiku), without whom the foundations of this work would not have existed.

As YADL is based on YAGO 3, we acknowledge the work made by its authors to prepare the original knowledge base, as well as their efforts into manually evaluating it.

REFERENCES

[1] Jens Bleiholder and Felix Naumann. 2009. Data fusion. *ACM computing surveys (CSUR)* 41, 1 (2009), 1–41.

[2] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 709–720. https://doi.org/10.1109/ICDE48307.2020.00067

[3] Sonia Castelo, Rémi Rampin, Aécio Santos, Aline Bessa, Fernando Chirigati, and Juliana Freire. 2021. Auctus: a dataset search engine for data discovery and augmentation. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2791–2794.

[4] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David R. Karger. 2020. ARDA: Automatic Relational Data Augmentation for Machine Learning. *Proc. VLDB Endow.* 13, 9 (2020), 1373–1387. https://doi.org/10.14778/3397230.3397235

[5] Alexis Cvetkov-Iliev, Alexandre Allauzen, and Gaël Varoquaux. 2023. Relational data embeddings for feature enrichment with background information. *Machine Learning* (2023), 1–34.

[6] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System.. In *Cidr*.

[7] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2022. Turl: Table understanding through representation learning. *ACM SIGMOD Record* 51, 1 (2022), 33–40.

[8] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. 2009. Truth discovery and copying detection in a dynamic world. *Proceedings of the VLDB Endowment* 2, 1 (2009), 562–573.

[9] Yuyang Dong, Chuan Xiao, Takuma Nozawa, Masafumi Enomoto, and Masafumi Oyamada. 2023. DeepJoin: Joinable Table Discovery with Pre-trained Language Models. *Proc. VLDB Endow.* 16, 10 (2023), 2458–2470. https://doi.org/10.14778/3603581.3603587

[10] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2021. COCOA: COrrelation COefficient-Aware Data Augmentation. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT*. OpenProceedings.org, 331–336. https://doi.org/10.5441/002/EDBT.2021.30

[11] Grace Fan, Jin Wang, Yuliang Li, and Renée J. Miller. 2023. Table Discovery in Data Lakes: State-of-the-art and Future Directions. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/3555041.3589409

[12] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1001–1012.

[13] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1190–1201.

[14] Javier Flores, Sergi Nadal, and Oscar Romero. 2021. Towards Scalable Data Discovery. https://doi.org/10.5441/002/EDBT.2021.47

[15] Sainyam Galhotra, Yue Gong, and Raul Castro Fernandez. 2023. Metam: Goal-Oriented Data Discovery. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2780–2793. https://doi.org/10.1109/ICDE55515.2023.00213

[16] César A. Galindo-Legaria. 1994. Outerjoins as Disjunctions. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 348–358. https://doi.org/10.1145/191839.191908

[17] Google. 2023. Dataset Search. https://datasetsearch.research.google.com/

[18] Leo Grinsztajn, Edouard Oyallon, and Gael Varoquaux. 2022. Why do tree-based models still outperform deep learning on typical tabular data?. In *Conference on Neural Information Processing Systems*.

[19] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing Google's Datasets. In *SIGMOD*. ACM, 795–806. https://doi.org/10.1145/2882903.2903730

[20] Madelon Hulsebos, Xiang Deng, Huan Sun, and Paolo Papotti. 2023. Models and Practice of Neural Table Representations. In *SIGMOD*. ACM, 83–89. https://doi.org/10.1145/3555041.3589411

[21] Viet-Phi Huynh, Yoan Chabot, Thomas Labbé, Jixiong Liu, and Raphaël Troncy. 2022. From Heuristics to Language Models: A Journey Through the Universe of Semantic Table Interpretation with DAGOBAH. In *Semantic Web Challenge on Tabular Data to Knowledge Graph Matching (SemTab)*.

[22] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String similarity joins: An experimental evaluation. *Proceedings of the VLDB Endowment* 7, 8 (2014), 625–636.

[23] James Max Kanter and Kalyan Veeramachaneni. 2015. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE international conference on data science and advanced analytics (DSAA)*. IEEE, 1–10.

[24] Aamod Khatiwada, Grace Fan, Roee Shraga, Zixuan Chen, Wolfgang Gatterbauer, Renée J. Miller, and Mirek Riedewald. 2023. SANTOS: Relationship-based Semantic Table Union Search. *Proc. ACM Manag. Data* 1, 1 (2023), 9:1–9:25. https://doi.org/10.1145/3588689

[25] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J. Miller. 2022. Integrating Data Lake Tables. *Proceedings of the VLDB Endowment* 16, 4 (dec 2022), 932–945. https://doi.org/10.14778/3574245.3574274

[26] Christos Koutras, George Siachamis, Andra Ionescu, Kyriakos Psarakis, Jerry Brons, Marios Fragkoulis, Christoph Lofi, Angela Bonifati, and Asterios Katsifodimos. 2021. Valentine: Evaluating Matching Techniques for Dataset Discovery. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE. https://doi.org/10.1109/icde51399.2021.00047

[27] Arun Kumar, Jeffrey Naughton, Jignesh M Patel, and Xiaojin Zhu. 2016. To join or not to join? thinking twice about joins before feature selection. In *Proceedings of the 2016 International Conference on Management of Data*. 19–34.

[28] Jiabin Liu, Chengliang Chai, Yuyu Luo, Yin Lou, Jianhua Feng, and Nan Tang. 2022. Feature Augmentation with Reinforcement Learning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE. https://doi.org/10.1109/icde53745.2022.00317

[29] Farzaneh Mahdisoltani, Joanna Biega, and Fabian Suchanek. 2014. Yago3: A knowledge base from multilingual wikipedias. In *7th biennial conference on innovative data systems research*. CIDR Conference.

[30] George A. Miller. 1994. WordNet: A Lexical Database for English. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*. https://aclanthology.org/H94-1111

[31] Fatemeh Nargesian, Abolfazl Asudeh, and H. V. Jagadish. 2022. Responsible Data Integration: Next-generation Challenges. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2458–2464. https://doi.org/10.1145/3514221.3522567

[32] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (2019), 1986–1989. https://doi.org/10.14778/3352063.3352116

[33] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table union search on open data. *Proceedings of the VLDB Endowment* 11, 7 (mar 2018), 813–825. https://doi.org/10.14778/3192965.3192973

[34] Phuc Nguyen, Ikuya Yamada, Natthawut Kertkeidkachorn, Ryutaro Ichise, and Hideaki Takeda. 2021. SemTab 2021: Tabular Data Annotation with MTab Tool. In *(ISWC (CEUR Workshop Proceedings, Vol. 3103)*. CEUR-WS.org, 92–101. https://ceur-ws.org/Vol-3103/paper8.pdf

[35] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D Lawrence. 2022. Challenges in deploying machine learning: a survey of case studies. *Comput. Surveys* 55, 6 (2022), 1–29.

[36] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[37] Liudmila Ostroumova Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: unbiased boosting with categorical features. In *NeurIPS*. 6639–6649. https://proceedings.neurips.cc/paper/2018/hash/14491b756b3a51daac41c24863285549-Abstract.html

[38] Vraj Shah, Arun Kumar, and Xiaojin Zhu. 2017. Are Key-Foreign Key Joins Safe to Avoid when Learning High-Capacity Classifiers? *Proc. VLDB Endow.* 11, 3 (2017), 366–379. https://doi.org/10.14778/3157794.3157804

[39] Junaid Shuja, Eisa Alanazi, Waleed Alasmary, and Abdulaziz Alashaikh. 2021. COVID-19 open source data sets: a comprehensive survey. *Applied Intelligence* 51 (2021), 1296–1325.

[40] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*. 697–706.

[41] Ritchie Vink, Stijn de Gooijer, Alexander Beedie, Marco Edward Gorelli, J van Zundert, Weijie Guo, Gert Hulselmans, universalmind303, Orson Peters, Marshall, chielP, nameexhaustion, Matteo Santamaria, Daniël Heres, Josh Magarick, ibENPC, Moritz Wilksch, Jorge Leitao, Jonas Haag, Marc van Heerden, cmdlineluser, Oliver Borchert, Chris Pryer, Ryan Russell, Joshua Peek, Colin Jermain, Adrián Gallego Castellanos, Jeremy Goh, and Liam Brannigan. 2024. *pola-rs/polars: Python Polars 0.20.6*. https://doi.org/10.5281/zenodo.10573881

[42] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. 2019. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*. 847–864.

[43] Eric Zhu, Vadim Markovtsev, aastafiev, Wojciech Łukasiewicz, Adam Foster, Sinusoidal36, Andrii Oriekhov, Joe Halliwell, JonR, Kevin Mann, Keyur Joshi, Peter Kubov, Qin TianHuan, Senad Ibraimoski, Spandan Thakur, Stefano Ortolani, Titusz, Vojtech Letal, Zac Bentley, fpug, hguhlich, long2ice, oisincar, and ronassa. 2023. *ekzhu/datasketch: v1.5.9*. https://doi.org/10.5281/zenodo.7654815

[44] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proc. VLDB Endow.* 9, 12 (2016), 1185–1196. https://doi.org/10.14778/2994509.2994534