Discovering Related Data At Scale

Sagar Bharadwaj
Microsoft Research
t-sabhar@microsoft.com

Praveen Gupta
Microsoft Research
t-pravgu@microsoft.com

Ranjita Bhagwan
Microsoft Research
bhagwan@microsoft.com

Saikat Guha
Microsoft Research
saikat@microsoft.com

ABSTRACT

Analysts frequently require data from multiple sources for their tasks, but finding these sources is challenging in exabyte-scale data lakes. In this paper, we address this problem for our enterprise's data lake by using machine-learning to identify related data sources. Leveraging queries made to the data lake over a month, we build a *relevance model* that determines whether two columns across two data streams are related or not. We then use the model to find relations at scale across tens of millions of column-pairs and thereafter construct a *data relationship graph* in a scalable fashion, processing a data lake that has 4.5 Petabytes of data in approximately 80 minutes. Using manually labeled datasets as ground-truth, we show that our techniques show improvements of at least 23% when compared to state-of-the-art methods.

PVLDB Reference Format:
Sagar Bharadwaj, Praveen Gupta, Ranjita Bhagwan, and Saikat Guha. Discovering Related Data At Scale. PVLDB, 14(8): 1392 - 1400, 2021.
doi:10.14778/3457390.3457403

1 INTRODUCTION

Analysis tasks frequently need multiple data streams [1] spanning different organizational groups within a data lake. For instance, to create a "collaboration graph" that links users who work with each other, an analyst has to process several streams from multiple collaboration platforms such as those providing email, video conferencing and instant-messaging. Another example is a task that determines the cause of service downtime and attributes it to either faulty application-level components, faulty network components, or malfunctioning hardware. This requires information from the application's various components or micro-services, the underlying network, and its compute infrastructure.

Analysts find it extremely laborious to discover such related sources of data in large data lakes, a task that takes them many days or even months. To make matters worse, data lakes are extremely large and continuously growing. Microsoft's data lake has roughly doubled in size every year for the last decade. Also, unlike relational databases where foreign-key relationships often capture related data, in large unstructured data lakes, data in isolated organizational silos seldom have inter-relationships that are explicitly defined.

To address this problem, previous work [9, 25] has proposed several techniques to build "data relationship graphs" that captures related data sources. Building such a data graph boils down to answering the question, "Given a column in a data stream, what are the related columns in other data streams in the data lake?". These techniques make significant advances towards answering this question. However they suffer from two shortcomings. First, they require a single sweep over the entire data which, given the exabyte-scale of our data lake, is prohibitive. Second, they propose a fixed set of metrics and, to define relevance, a user of the data graph has to craft ways of combining them. Ideally, the definition of relevance should be specific to a data lake, and the data graph construction algorithm should automatically determine how to combine different metrics to give one formula for relevance.

While the immense scale of our data lake poses a daunting challenge, it also provides two unique opportunities. First, Cosmos, Microsoft's Data Lake, sees about one million jobs every day, each of which may run multiple `JOIN` queries on the data. Such large numbers enable us to treat *queries as data*. Using `JOIN` clauses they hold, we build a *relevance model* that captures a definition of relevance that is specific to the data lake. Second, Cosmos holds around two billion data streams with twenty six billion data columns, yielding an unprecedented amount of metadata. This allows us to treat *metadata as data* and we use metadata-specific features to build the relevance model. These features can capture related columns with names such as `machine` and `datacenter` which data-based features cannot. We also use data-based features inspired by previous-work [9, 25] but only on samples of data for textual columns.

In this paper, we propose the Data Lake Navigator(DLN), a system that builds and uses the relevance model to construct a data graph for Cosmos.

Our paper makes the following novel contributions:

- We use machine-learning on queries made to data streams, specifically `JOIN` clauses, to learn characteristics of related data columns.
- We use two metadata-based features: *embedding-enhanced column-name similarity* and *column-name uniqueness*. We use data-based features as well, but to scale well, instead of calculating them on full data, we base them on data samples.
- We show that a metadata-only approach to build data graphs is indeed feasible, provides value, and is scalable. We also show that data-based features do provide value, though more marginal than expected.
- We evaluate the relevance model for one large service, namely Office365 Core. Additionally, we build a data graph using this model for one of Microsoft's internal services and evaluate, using a manually labeled dataset, how well it detects useful relations not seen before in any queries.

We evaluate a *metadata-only* relevance model, which learns characteristics of related columns using only metadata-based features, and an *ensemble* model which uses metadata features for numeric columns, and both metadata and data-based features for textual columns. Our results, quite counter-intuitively, show that

Proceedings of the VLDB Endowment, Vol. 14, No. 8 ISSN 2150-8097.
doi:10.14778/3457390.3457403

[1] A data stream is analogous to a table.

the metadata-only model which uses only 2 features detects related columns with 0.96 precision and 0.92 recall, whereas the ensemble model improves this marginally to 0.97 and 0.95 respectively. Our approach also scales well, as we show that we can process a large data lake with 137,000 streams, 2.6 million columns, 30,754 unique columns and 4.5 Petabytes of data in 80 minutes. Furthermore, we compare our approach with three state-of-the-art techniques using a manually-labeled dataset. We show that DLN shows an improvement of at least 23% in F1-score over the state-of-the-art. We also show through several examples that DLN learns many interesting relationships which were not observed in any JOIN clauses.

2 BACKGROUND

In this section, we provide a brief overview of Cosmos, Microsoft's big-data processing system, and Scope, the language for processing data on Cosmos. For a more in-depth description of Cosmos and Scope refer to [4, 24].

Cosmos [4] stores multiple exabytes of mostly telemetry data and is used daily for analytical jobs by every product team at Microsoft. Data in Cosmos is stored in either unstructured or structured streams. Unstructured streams, similar to files in a file-system, are stored as opaque byte streams where Cosmos is oblivious to any metadata. Structured streams, similar to tables in a database, are stored as rows or columns with additional schema metadata. Cosmos offers more efficient APIs for extracting data from structured streams. Unlike file-systems and databases, Cosmos data is stored in very large pages, called *extents*, that can be 1Gb or more in size. Extents are compressed and encoded for optimized sequential access at the cost of random access.

The design of Cosmos has resulted in some best-practices when dealing with log data and telemetry. Raw data, such as web-server logs, is initially ingested and transformed into a structured stream. Another script then enriches it by adding additional useful columns, for example joining with other reference data and computed columns, to produce a final (cooked) stream that other teams can consume. The cooked stream is structured to enable efficient querying.

Data is processed and consumed using Scope [24] scripts. Scope is a SQL dialect to filter, join, and select from data stored in Cosmos. A user or service submits a Scope script to the Cosmos cluster, which then compiles and executes the job, and if the job completes successfully, persists the output back into Cosmos. The Scope compiler stores various compile-time artifacts including the submitted script and generated query plan for debugging and later analysis. In Section 4.1 we describe how we parse this generated query plan to create ground-truth for our models.

Teams often provide multiple *Views* of their streams. A View, similar to a (non-materialized) SQL view, generates a logical dataset at query time by transforming an underlying physical dataset. This ensures forward compatibility of downstream consumers to breaking schema changes in the underlying stream. A secondary reason is to include computed columns or joins that are too expensive to store separately. The Scope compiler inlines Views at compile time such that the generated query plan includes joins and other operators from Views and the calling script alike.

2.1 Scaling Challenges

Cosmos stores multiple exabytes of data in billions of unstructured and structured streams. The streams have 13 columns on average, though the maximum encountered column count exceeds 8000. Row counts typically exceed tens of millions for cooked streams. Over 1 million Scope jobs are processed by the cluster on any given day. These jobs are submitted by over 5000 users, and several thousand service accounts.Given this scale, metadata access is significantly faster than data access because of the following four reasons.

First, metadata requires reading only the metadata block from disk, and in newer versions is stored in-memory in a distributed service obviating any disk access. Thus a data discovery approach that works in metadata-only mode will be significantly faster today and more so in the future.

Second, data access in Cosmos is optimized for batch-processing throughput and not interactive latency. As such, an implicit assumption is that Scope scripts will access a small number of streams (few tens) and typically consume rows sequentially. This assumption is borne out in the vast majority of production workloads. However, building a data graph entails a dramatically different access pattern, one that spans a large number of streams. Cosmos does not support random sampling either. We do not envision Cosmos being optimized for our workload and so pick a design point that trades off some correctness for performance, i.e. we use metadata predominantly, and sample only the top 1000 rows as data. Section 4.2 further quantifies the costs that drove this decision.

Third, Views present an additional performance challenge. Since Views are arbitrary pieces of Scope code, they may hide expensive joins and other computations. For metadata-only approaches, Views present no cost since the schema of the View is declared in the View code and available at compile-time. Sampling data from a View, however, requires materializing the entire view. Retrieving even a single row from a view may involve a resource intensive computation (e.g. aggregation).

Finally, in addition to performance implications, there are compliance and audit implications for accessing data. Accessing data has a significantly higher compliance workload than accessing metadata. Given the ease of gathering metadata, we have given particular attention to developing an accurate metadata-only model.

3 PROBLEM AND SOLUTION OVERVIEW

We address the problem of building a data graph for Microsoft's data lake. We break this down into two parts. First, we build a relevance model that captures relevance between column-pairs in the data lake. Second, we design scalable algorithms to build the data graph using this relevance model.

3.1 Building the Relevance Model

Previous work[3, 25] has used several metrics for relevance, such as number of overlapping values in columns, content similarity and schema similarity. We believe that a more appropriate way to capture relevance is to *learn a combination* of all such metrics, *i.e.* it is better learned using examples of column-pairs that are already known to be related. Fortunately, we have access to a large number of Scope queries that have run on Cosmos in the past. We believe that the JOIN clauses in these queries give us a rich dataset of related columns that captures the collective knowledge of a large number of developers and data analysts across a diverse set of streams. We therefore build a relevance model, which is a machine-learning classifier that is trained on such column-pairs, using the following steps:

Parsing queries: For each Scope query, the compiler generates a query plan. Our query plan parser discovers JOIN clauses, each of which yields at least one related column-pair. We label these pairs as positive samples for learning the classifier. To determine negative samples, we randomly choose column-pairs from two randomly chosen streams which we have not seen occurring together in any JOIN clauses. Section 4.1 provides more details on the parser.

Extracting Features: For every sample column-pair, we extract *metadata-based features* using Cosmos's metadata store and *data-based features* calculated over the top 1000 rows of a column. Section 4.2 describes these features.

Machine learning: Using the positive and negative samples and their features, we then use a random-forest classifier that learns whether two columns are related or not. The model also provides a *relevance probability*, which captures the degree of the relevance. We learn three classifiers: a *metadata-only* classifier which uses only metadata-based features, *data-only* classifier that uses only data-based features for textual columns, and an *ensemble* classifier that uses metadata-based features for numeric columns and both metadata and data-based features for textual columns. We use the data-only classifier only for evaluation and comparison. Section 4.3 describes the learning process in detail and Section 6.1 describes our evaluation and comparison of these models.

3.2 Building the Data Graph

We use the following process to build the data graph. For each column-pair, the relevance model determines if the pair is related or not, and what the relevance probability is. If the relevance probability is higher than 0.5, we add an edge in the data graph, setting the edge weight to the relevance probability. Considering every single column-pair is prohibitively expensive. We therefore prune the data using two techniques: metadata clustering, and reverse-index mapping. Section 5 describes our pruning approach while Section 6.2 shows that the data graph creation algorithm scales well.

4 BUILDING THE RELEVANCE MODEL

In this section, we describe each step towards building the relevance model: parsing queries, extracting features, and machine-learning.

4.1 Parsing Queries

Given a script's query plan, the parser extracts column-pairs from JOIN clauses [2] where each JOIN statement can have multiple clauses. Figure 1 shows an example representation of a query plan. The nodes in the graph represent data streams and the directed edges represent operators that transform streams. The black nodes or *input nodes* denote input data streams and the grey node, an *output node*, shows the output data stream. The parser identifies JOIN operators (shown by bold arrows in the figure) and traverses the tree back from these operators to the input node to discover which data stream columns are being joined.

[2]We ignore self-joins

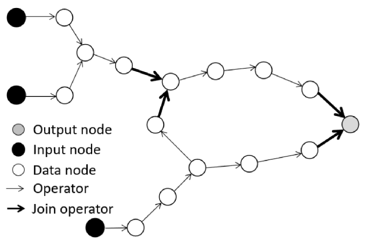

Figure 1: Query plan of a Scope Script.

Parsing query plans poses several challenges which could lead to imperfections and loss in ground-truth data. We describe three such challenges and how we address them.

Aliasing: One of the main causes of imperfection is aliasing of column names. For instance, a column may be renamed in a User Defined Function (UDF) and the JOIN may happen on this renamed column. The Scope compiler does not parse UDFs, hence the query plan loses information that ties output columns from the UDF to input columns. We alleviate this problem by choosing the most likely input column for a given output column using string edit distance between the column names. In other words, we tie an output column from the UDF to the input column that has the *closest* name to it. This makes the simplifying assumption that an output column is related to just one input column. Any case where such an assumption does not hold, contributes noise to the ground-truth.

Semantic relationships: Simple features may not capture information about columns that are related at a semantic-level. For instance, a JOIN clause equated a column of IP addresses with another that contained host names, albeit with appropriate transformations. Data-based features such as inclusion dependence will not capture such relationships. To compensate, we use word embeddings pre-trained on software domain data [10] that help us capture such complex semantics in the column names of such pairs (details in Section 4.2).

String transformations: A JOIN clause may apply string-based transformations on content. For instance, it could involve equality between the whole string contained in one column and a substring of values in another column. Again, data-based features will not capture such similarities without applying the same substring function on the appropriate column. We have to rely entirely on metadata to capture this similarity. While there is literature available on similarity joins [6, 14, 15] and fuzzy string matching [5, 21] we do not resort to these methods as column metadata captures many such relationships and keeps computation cheap.

For our experiments, we concentrate on parsing queries run on the Office365 Core service's data. We have parsed 1.6 million scripts written and run over a period from Aug 01, 2020 to Aug 31, 2020. Parsing 1.6 million Scope scripts completed in under 6 hours. Figure 2 gives an idea of number of JOIN statements per script we parsed. While most of the scripts have fewer JOIN statements, the tail is long and the maximum number of joins per script was 120. We see a spike at 11 joins because of a large cluster of 103K jobs performing the same joins across multiple dates.

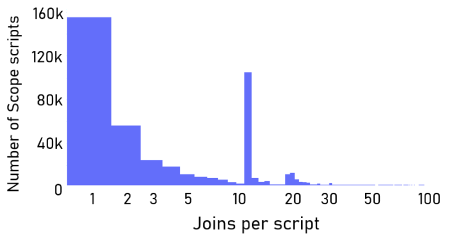

Figure 2: Histogram of Join statements per script.

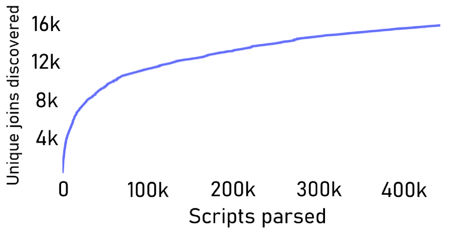

Figure 3: Fraction of unique join pairs discovered against fraction of scripts parsed.

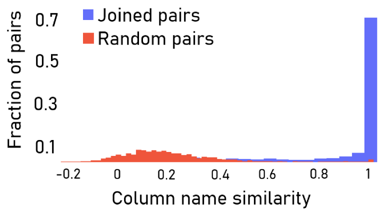

Figure 4: Distribution of column name similarity.

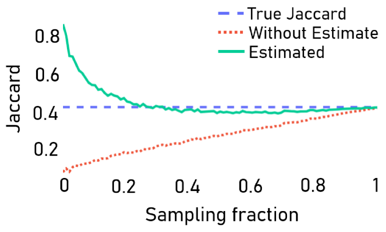

Figure 5: Behaviour of Jaccard similarity estimate for a pair of columns with a highly skewed distribution.

While we parse a large number of scripts, we notice that the number of unique column-pairs is only about 16,000, i.e. we see continuously repeated `JOIN` clauses. Figure 3 shows that about 90% of unique pairs are covered by 55% of the scripts we parse. This is because several scripts run repeatedly as daily jobs. Also, analysts write queries repeatedly using relations they are already aware of. This further makes the case for automated discovery of related data. It also motivates us to perform rigorous feature engineering which we describe in Section 4.2.

4.2 Extracting Features

In this section, we first describe the metadata and data-based features that our model uses. Next, we describe experiments that quantify the performance of extracting sampling-based data features.

4.2.1 Metadata-based Features. Cosmos has a metadata store which captures all stream and column metadata. We extract a total of 4.2 million different column names for the Office365 Core service. For all column names, we tokenize the name by using popular naming conventions such as camel-case and snake-case. For example, if the column name is `machinename_backend`, it splits into two tokens: `machinename` and `backend`. Unfortunately, we observe that such clean naming conventions are not rigorously followed and hence in addition we use a word segmentation algorithm based on a large corpus [17] to further subdivide the tokens into recognizable English words wherever possible. For example, the token `machinename` further divides into tokens `machine` and `name` using the word segmentation algorithm.

For each token thus obtained, we compute the *Inverse Term Frequency* (ITF). Given a token t and the number of column names it appears in, say n_t, and the total number of column names N, the ITF for the token is calculated as $ITF_t = \log(\frac{N}{n_t})$. The ITF captures how commonly used the token is across all column names: the larger the ITF, the less common the token. We use the tokens and the ITF to compute two metadata-based features for each column-pair:

Column Name Similarity: Semantic similarity between column names can indicate relevance. Therefore, we use column name similarity as a feature. To capture column name similarity, we first tokenize the two column names as explained above. Next, we use word embeddings trained on software domain data [10] to get a vector representation of each token. Word embeddings capture related column names that may not have character level similarity but refer to similar content. For instance column names such as `MachineName` and `Server` refer to the same entity. The vector that we assign to each column name is a weighted mean of individual token vectors, where the weights are the token's ITF. This ensures that the column vector is dominated by the vector representation of an uncommon token. Also, common terms such as `id`, `date` and `time` contribute very little to the similarity score. Finally, we calculate the similarity of a column-pair as the cosine distance between their assigned vectors. Figure 4 shows the variation in distribution of column name similarity between randomly selected pairs and pairs from the parsed `JOIN` clauses. There is a clear separation in these values, as most of the columns that are joined in the data lake have names with a similarity score >0.9.

Column Name Uniqueness: A column name is as unique as its most unique token. Given a column name, we set its uniqueness score to the maximum ITF of all its individual tokens. We use the uniqueness scores of both column names as two separate features.

4.2.2 Data-based Features. We calculate data-based features using the top 1000 values in a column for the reasons explained in Section 2.1. We consider data-based features only for textual columns as we found that numeric values repeat very often across unrelated columns thereby leading to false-positives. For instance, a column containing error codes 0 to 4 has identical values as a column containing function return values, also 0 to 4.

Jaccard Similarity: The fraction of common values between two columns indicates how related they are. Hence, we use Jaccard similarity as one of the features. Jaccard similarity between two columns is the ratio of the size of their intersection and the size of their union. Since we are working with samples, it is not possible to calculate exact Jaccard similarity and hence we use a novel approximation method.

Let A and B be the sets of all distinct values in the columns a and b respectively. True Jaccard similarity is the ratio of $|A \cap B|$ and $|A \cup B|$. $|A|$ and $|B|$ can be estimated using Shlosser's distinct value estimate [20]. However, since we only have access to distinct values within our samples, say A' and B', we need to apply a transformation on these to achieve an estimate of the true Jaccard Similarity. Let $p_a = \frac{|A'|}{|A|}$ and $p_b = \frac{|B'|}{|B|}$, which are approximations for the fraction of distinct values sampled from a and b respectively. We estimate Jaccard similarity as:

$$J(a,b) = \frac{\frac{|A' \cap B'|}{p_a p_b}}{\frac{|A'-B'|}{p_a} + \frac{|B'-A'|}{p_b} + \frac{|A' \cap B'|}{p_a p_b}}$$

This is based on the assumption that $|A \cap B|$ can be approximated to $\frac{|A' \cap B'|}{p_a p_b}$, as the probability of a distinct value being sampled from both the columns a and b is $p_a p_b$. We observe that this gives us larger than true Jaccard estimates for skewed distributions and lower estimates for uniform distributions. Figure 5 shows how the estimate behaves for a pair of highly skewed columns in the Office365 Core dataset, with a squared coefficient of variation of value frequencies (γ^2) equal to 727. Note that $\gamma^2 = 0$ would mean all values occur equally frequently in the column. The "Without Estimate" trace in the graph is the value of $\frac{|A' \cap B'|}{|A' \cup B'|}$, i.e. the Jaccard Similarity calculated on samples. "True Jaccard" is $\frac{|A \cap B|}{|A \cup B|}$ while "Estimated" is Jaccard similarity score estimated as shown above. We defer a theoretical analysis of the consequences of our simplified assumptions to future work. The empirical results show that the estimated Jaccard Similarity can sufficiently represent similarity.

Inclusion Dependence: Some column pairs may have a low Jaccard similarity and yet one column may entirely contain the other. This is akin to pairs that share a primary key-foreign key relationship in a relational database. To take such pairs into consideration, we also include inclusion dependence as a feature, where the true score of inclusion of a in b is $\frac{|A \cap B|}{|B|}$. Similarly, true score of inclusion of b in a is $\frac{|A \cap B|}{|A|}$. However, we only have A' and B' which are distinct values in the samples of columns a and b. We estimate inclusion dependence of a in b as:

$$I(a_b) = \frac{\frac{|A' \cap B'|}{p_a p_b}}{\frac{|B'-A'|}{p_b} + \frac{|A' \cap B'|}{p_a p_b}}$$

. Inclusion dependence of b in a, $I(b_a)$, can be similarly estimated. We use $I(a_b)$ and $I(b_a)$ as features. The assumptions and intuition behind the Jaccard Similarity estimate carry over to Inclusion dependence estimates as well.

Pattern Similarity: Given we are working with samples, the size of intersection between the samples of columns with a very high cardinality can be 0 in spite of them being related. For example, all Globally Unique Identifiers (GUIDs) are 128-bit alphanumeric identfiers grouped according to a pre-determined standard. Two columns containing GUIDs may be related but samples will probably not have common values. To capture such similarities, we mask all hexadecimal characters in the samples and estimate the Jaccard similarity score between these masked strings.

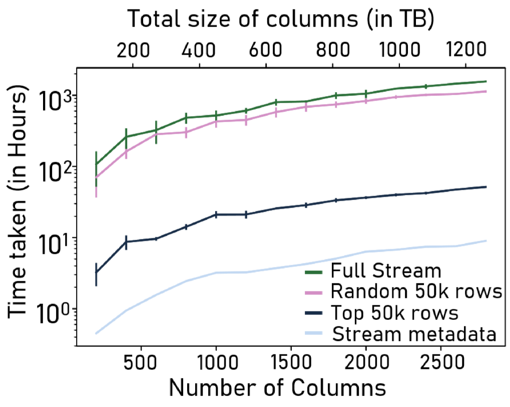

Figure 6: Performance comparision for scanning full column, randomly sampling, top sampling and crawling for metadata.

4.2.3 Performance of Feature Extraction. Figure 6 explains the performance difference between scanning full columns, randomly sampling from the columns, scanning the top 50k values from columns, and performing a metadata-only crawl. Note that the median number of entries in a column is extremely high, roughly 500 million. As can be seen, full scans and random sampling are both prohibitively expensive, at 1984 seconds/column and 1490 seconds/column respectively. Random sampling is expensive because to get a good sample, all extents of the column have to be read (Section 2.1 provides context on this). Sampling the top 50k rows on the other hand takes only 66 seconds/column. Clearly, the metadata-only model scales the best given that crawling metadata takes only 8 seconds/column and is 8 times faster than top sampling.

4.3 Machine Learning

The final step in building the relevance model is to learn the classifier given the positive and negative samples. We tried 10 different classifiers, including linear classifiers like logistic regression and non-linear classifiers such as random forests[2], gradient boost[13] and AdaBoost[12]. While the non-linear classifiers worked significantly better than linear classifiers, they all performed similarly, with the random forest-based classifier using an ensemble size of 100 trees giving marginally better results than others. Given a column-pair, the classifier outputs a fraction between 0 and 1: if this value is above 0.5, the columns are related. We call this the relevance probability. The higher the probability, the more related a column-pair.

We learn two models: a metadata-only model which uses column name similarity and column name uniqueness as features and an ensemble model that uses metadata features for numeric columns and all features described for textual columns.

5 BUILDING THE DATA GRAPH

Algorithm 1 Data graph construction algorithm

```
Metadata Crawling and Data Sampling
    $n \leftarrow$ List of columns to use as nodes of the graph
    for Column $c \in n$ do
        Sample $s$ values in $c$
        Fetch and store metadata for $c$
Create reverse indices
    $X \leftarrow$ Set of distinct data samples
    for Sample $x \in X$ do
        Map $x$ to list of columns containing $x$
Prune and generate candidate column pairs
    K-means clustering with column name embeddings
    Generate all column pairs within each cluster
    Generate all column pairs with overlap in sampled data
Calculate Features and Predict related column pairs
    for Pair $p \in$ Candidate Pairs do
        Calculate metadata and data features
        Use the pre-trained classifier to classify based on selected threshold of probability
```

In this section we describe the data graph construction algorithm, which uses the relevance model as a building-block. The data graph is built in three steps. The first step, feature extraction, is the same as described in Section 4.2. In the *pruning* step, rather than consider every column-pair which would be prohibitively expensive, we find a reduced set of candidate column-pairs to input to the relevance model. In the *discovery* step, we use the model to discover related column-pairs and add edges to the data graph. Algorithm 1 summarizes the process. In the worst case, the prediction algorithm needs to be executed on $O(n^2)$ pairs of columns. However, the pruning steps significantly cut down the number of candidate pairs.

5.1 Pruning and Discovery

We adopt a two-pronged approach to pruning column-pairs, metadata-based and data-based pruning.

Metadata Clustering. We use k-means clustering on column name embeddings to detect similar columns. We use a total of 40 clusters. Any two columns that lie in the same cluster form a candidate column-pair for the data graph. We eliminate any column-pair which span two different clusters.

Reverse index-based Pruning. There is extensive literature on content-based pruning using top k set similarity search [1, 6, 11, 25]. However, since we are working with samples, we do not have the liberty to set a threshold on k, as even a single intersection may be valuable. The size of intersection between two samples is not necessarily proportional to how related they are. For example, samples from two columns representing IDs of the same entity may have very little intersection because of its huge cardinality. Hence, we form a reverse index on all the samples we collect, as the number of unique values in the content is bound by number of samples $\times$ number of columns. For a given data value, the reverse index holds all the columns that hold that value. From the reverse index, we prune all column pairs that have no value in common. We consider all other column-pairs as candidates for the data graph.

Table 1: Metrics evaluating the relevance model.

Metric	metadata-only	data-only	ensemble
Precision	0.96	0.96	0.97
Recall	0.92	0.82	0.95
F1-Score	0.94	0.89	0.96

The set of candidate column-pairs is the union of the outputs of both pruning techniques. Eliminating column pairs that have no value in common may remove some related columns whose samples do not have an intersection. We evaluate this on our datasets in Section 6.2.2.

In the discovery phase, we input the column-pairs detected by pruning to the relevance model and finally get a list of related column-pairs, or edges in our data graph. Each edge is also annotated by the relevance probability.

6 EVALUATION

We use the `sklearn`[18] toolkit in Python to learn the random forest model and evaluate it. The data graph construction algorithm, including pruning and discovery, is written in Scope and runs on Cosmos. We import the relevance model into our Scope script using support that Scope provides for Python UDOs. To achieve scale and reduce latency, we have parallelized several steps of data graph construction.

For querying the data graph, we use a model very similar to Aurum [3]. We have currently built a search engine-like interface that takes as input a column name and outputs a ranked list of related columns. We also provide a tree-map visualization which captures the distance in folder hierarchy between the related columns.

Our evaluation answers questions in two categories. We first evaluate *Relevance Model Accuracy* or how well the relevance model captures relationships in `JOIN` clauses. We next evaluate *Data Graph Efficacy*, where we evaluate the performance of our data graph construction algorithm, and we measure how well the data graph captures relationships that have not been seen before in `JOIN` clauses.

6.1 Relevance Model Accuracy

Dataset: We use Office365 Core queries to evaluate the relevance model. This data lake contains telemetry about service health, such as CPU and I/O usage, and event logs from micro-services. We obtained 4045 column-pairs from past joins seen in Office365 Core's query history as positive samples. To balance the dataset, we chose 4045 column-pairs at random as negative samples.

We evaluate the efficiency of the metadata-only, data-only and the ensemble models by performing cross-validation: we use 80% of `JOIN` clauses for training and 20% for testing. Note that this evaluation uses only `JOIN` clauses, and does not consider column-pairs that have so far not been observed in `JOIN`s. We evaluate the latter in Section 6.2.

Table 1 shows the precision, recall and F1-score for the metadata-only, data-only and the ensemble models. The metadata-only model performed surprisingly well with 0.96 precision and 0.92 recall, while adding the data-based features for the textual columns in the ensemble model improved the precision to 0.97 and recall to 0.95. The data-only model for textual columns does not perform as well

Table 2: Metric improvement with each added feature in the metadata-only model.

Metric	equality	embedding	+uniqueness
Precision	1.00	0.88	0.96
Recall	0.65	0.90	0.92
F1-Score	0.79	0.89	0.94

as the metadata-only model with recall 0.82 showing that metadata features are absolutely necessary to solve this problem at scale.

To investigate the metadata-only model further, we explored how each feature we use improves the metrics. We started with a baseline model with only one feature, *column name equality*. In other words, the baseline model says a column-pair is related if and only if the column names are exactly the same. The results are shown in Table 2. Column name equality is able to capture 65% of all related column-pairs in our training dataset (recall=0.65). Not surprisingly, the precision is very high. When we use column name similarity, the recall shoots up to 0.90, but the precision drops to 0.88. Adding column name uniqueness brings the precision back up to 0.96. These results show that there is value added by each one of the metadata features. Note that to improve precision of the system, we can just consider column name equality. Columns that have exactly same names are related in most cases, resulting in high precision. However, this would come at the cost of missing interesting relationships between columns that do not share a name, thereby reducing recall drastically.

We now provide some examples of column-pairs in our test-set to explain the benefits of metadata features and data features.

TP in metadata-only, FN in data-only: When related column-pairs contain randomly generated GUIDs, the columns have a very high cardinality. Their samples may not have any common values. In such examples, data features such as Jaccard similarity and Inclusion dependence will be estimated as 0. However, in most such cases, they have very similar column names which will be captured by the column name similarity feature. For example, two columns named `user_id` and `CustomerId` that are related is captured by embedding-enhanced column name similarity.

TP in data-only, FN in metadata-only: Often, users use shortened column names which metadata features do not capture. For example, two columns named `ManagerName` and `mngr` may be related, but only the data based features are able to capture the similarity.

TN in data-only, FP in metadata-only: Two columns with the exact same name may refer to completely different formats of data. For example, two columns both of which are named `CommitId` may appear related, but they refer to commits made to two code repositories and have different formats. Hence they refer to different entities and are not related.

6.2 Data Graph Efficacy

In this section, we first evaluate the performance of algorithms that we use to construct the data graph. Next, we evaluate how well the data graph captures relations and compare our techniques to three state-of-the-art data discovery techniques [3, 8, 25].

6.2.1 Performance. To evaluate performance of data graph construction at scale, we use the data lake of the service that supports DevOps within Microsoft. This data lake stores information of every git repository Microsoft uses and associated code commit, build and test information. It has 137,000 streams and a total of 2.6 million columns. However, many of these streams contain similar information and have the same schema, but are recorded on different dates. We remove such repetitions and reduce our analysis down to 30,754 unique columns adding up to 4.5 Petabytes in size.

30,754 columns in the evaluation dataset can lead to more than 450 million column-pairs. Our pruning algorithms reduced this to approximately 40 million column-pairs. Using the relevance model on these, we built a data graph with 12 million edges that had a probability > 0.5. 2 million of these edges had a probability > 0.9.

Table 3 shows the time used to build the data graph for this dataset. The duration gives the end-to-end time required for each step, while the compute time gives the total time across all parallel components for that step. We show that creating the data graph for this data lake took only 81 minutes (4861s) and the total compute time was 216 hours (775890s). This shows that our pruning and discovery techniques do indeed scale well to large data lakes.

Here are some interesting examples of columns we found were related which were not present in the training data. Two columns named `BugID` and `IssueID` were marked as being related and both columns referred to the ID of a software bug reported on a bug tracking system. `ChangedDate` and `ModifiedDate` columns were marked as related with probability 1 due to semantic similarity captured by word vectors. `CommitSHA` and `MergeSourceCommitID` were marked as being related with probability 1 as well.

6.2.2 Accuracy. We now evaluate DLN's accuracy and compare it with state-of-the-art techniques. Previous work on data discovery focuses mostly on data based methods. JOSIE [25] finds the top K sets with highest overlap for a given query set. Aurum [3], which the Data Civilizer [9] uses, builds a linkage graph with edges like content similarity, schema similarity and primary key-foreign key relationships between columns. We compare DLN's ensemble model to both JOSIE and Aurum and the metadata-only model to a

Table 3: Performance of our implementation.

Stage	**Duration (s)**	**Compute time (s)**
Metadata crawling	800	49136
Data sampling	259	102525
Reverse index creation	44	44
Data-based Pruning	118	118
Metadata-based Pruning	32	32
Feature Extraction	3306	621888
Model Prediction	302	2147
Total	**4861**	**775890**

Table 4: DLN comparison with previous-work.

Metric	DLN Ensemble	DLN Metadata	PMI	Aurum Ensemble	JOSIE
Precision	0.92	0.73	0.42	0.75	**0.95**
Recall	**0.66**	0.45	0.55	0.42	0.24
F1-Score	**0.77**	0.55	0.47	0.54	0.39
Time (sec)	70.40	3.89	2.81	26.08	121.54

strawman model that uses Pointwise Mutual Information (PMI) [8] to determine similarity between columns. We used the code available for both JOSIE and Aurum and ported it to run on our system and data. We built the PMI-based strawman ourselves.

Dataset: To perform this evaluation, we selected 15 tables from Office365 Core service, containing 437 columns. We asked domain experts to manually label randomly chosen column-pairs from this data. Of this set, they labeled 266 column-pairs as related. 173 of these were textual column-pairs and the remaining 93 were numeric column pairs. We use the top-1000 sample for each column as input to JOSIE, Aurum, and DLN's ensemble model.

For Aurum, we used schema similarity edges for numerical columns and both schema and content similarity edges for textual columns. For JOSIE, we selected the top 5 columns with maximum overlapping values for each column, with a suitable threshold of overlap. We selected all required thresholds so as to maximize the F1-score for each approach.

To build a PMI-based baseline, we first tokenized all pairs of column names seen in the training set as described in Section 6.1. We count how many times a token-pair occurs across the two column names in a column-pair. We do this for all token-pairs. We use these counts to calculate PMI values for all token-pairs.

$$PMI(x,y) = \log\left(\frac{P(x,y)}{P(x) * P(y)}\right)$$

$$P(x,y) = \frac{Count\ of\ co\text{–}occurences\ of\ (x,y)}{\sum_{(a,b)} Count\ of\ co\text{–}occurences\ of\ (a,b)} \text{ and}$$
$$P(x) = \frac{Count\ of\ occurences\ of\ x}{\sum_a Count\ of\ occurences\ of\ a}$$

For a given column-pair, we generate all token-pairs, calculate the average of PMI values for these pairs, and classify the pair as related if this average is higher than a suitable threshold that we calculate through a validation phase.

Table 4 shows a comparison of all techniques. We note that the DLN's metadata-only model (F1-Score=0.55) performs better than all three previous techniques, and DLN's ensemble model (F1-Score=0.77) significantly out-performs them. We believe our use of the relevance model not only captures various modes of similarity, but it also captures several domain-specific relations which are left undetected without the use of machine-learning. The table also shows the total time taken to build the data graph in each approach.

Loss in recall due to pruning: Although the pruning methods described in section 5 help us scale our approach to large data lakes, they may remove related column-pairs, and cause a loss in recall. We evaluated the loss in recall due to pruning. We first calculate recall by generating all possible column-pairs in the dataset. Then, we repeat the same experiment by using the pruned set of column pairs as the candidate set. We see a decrease in recall from 0.68 to 0.66 due to pruning.

7 DISCUSSION AND FUTURE WORK

Our techniques depend upon JOIN clauses and queries being available to train the relevance model. However a relevance model trained on one dataset can potentially be used to construct a data graph for another dataset.

We believe this will be true if the datasets are similar, e.g. different data lakes within the same organization. To evaluate the feasibility of this approach across very different datasets, we built a generic relevance model with the dataset described in Section 6.1 and used it to construct a data graph for the public ChEMBL dataset [22] which is a well known chemical database of bioactive molecules. The ground-truth labels for this dataset are based off the PK-FK pairs in the SQL data dump.

DLN obtains a precision of 0.21 and a recall of 0.86. While some amount of loss in precision is expected given the different training dataset, the low precision is also due to a ground truth that is limited to known PK-FK pairs. For instance, DLN found the related column-pair `record_id` and `drug_record_id` which was not in the ground-truth. We find these results promising, and we leave a further analysis of such cross-domain testing to future-work.

Previous work has looked at discovering related data from slightly dissimilar data [5, 14, 15, 21]. This is useful when transformations are used before the JOIN clause. We intend to augment our techniques using these as well.

8 RELATED WORK

Data Civilizer [9] discovers related data in a two-step process. It first profiles available datasets to extract datatypes, cardinality and a column signature to build a linkage graph with lightweight relationships like content similarity and schema similarity. This is used for pruning search spaces for calculating expensive heavier relationships like inclusion dependency and structure similarity. It also finds primary key-foreign key relationships using past work on data based approaches for foreign key discovery [19]. Although it finds relationships between data nodes using several similarity measures as mentioned above, it needs a full pass on data columns for calculating the heavy relationships.

JOSIE [25] is a top K overlap set similarity algorithm for finding joinable columns. It uses an inverted dictionary which maps every distinct token in a data lake to a list of columns containing it. This is used to find columns with maximum intersections with the values of a query column. Although the work discusses ways of minimizing the cost of reading the dictionary to find these top K columns, it still needs a pass over the entire data in the search space to create this dictionary. It also ignores columns with numerical value as it can lead to a large number of unique tokens in the datalake. This can result in risk of losing interesting columns like `id`, `key` which can be in numerical formats.

Several efforts use data-based similarity detection techniques [7, 16, 23, 26] for finding relations in a corpus of heterogeneous tables. However, they require full scans of data which, as we have argued, is not feasible at scale.

9 CONCLUSION

In this paper, we have described a methodology that uses machine-learning to build a data graph that links related data in large data lakes. Our evaluation shows that using lightweight metadata-level features, we can build accurate models which can be used to find several kinds of related data sets. We also show that our techniques perform significantly better than the state-of-the-art techniques on Microsoft's data lake.

REFERENCES

[1] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up All Pairs Similarity Search. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. Association for Computing Machinery, New York, NY, USA, 131–140. https://doi.org/10.1145/1242572.1242591

[2] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[3] R. Castro Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. 2018. Aurum: A Data Discovery System. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, Paris, France, 1001–1012.

[4] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1265–1276. https://doi.org/10.14778/1454159.1454166

[5] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. 2003. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. Association for Computing Machinery, New York, NY, United States, 313–324.

[6] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A primitive operator for similarity joins in data cleaning. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, IEEE, Atlanta, GA, USA, 5–5.

[7] P. H. Chia, D. Desfontaines, I. M. Perera, D. Simmons-Marengo, C. Li, W. Day, Q. Wang, and M. Guevara. 2019. KHyperLogLog: Estimating Reidentifiability and Joinability of Large Data at Scale. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 350–364.

[8] Kenneth Ward Church and Patrick Hanks. 1990. Word Association Norms, Mutual Information, and Lexicography. *Computational Linguistics* 16, 1 (1990), 22–29. https://www.aclweb.org/anthology/J90-1003

[9] 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 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, Chaminade, California. http://cidrdb.org/cidr2017/papers/p44-deng-cidr17.pdf

[10] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. 2018. Word Embeddings for the Software Engineering Domain. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 38–41. https://doi.org/10.1145/3196398.3196448

[11] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set similarity joins on mapreduce: An experimental survey. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1110–1122.

[12] Yoav Freund and Robert E Schapire. 1997. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *J. Comput. Syst. Sci.* 55, 1 (Aug. 1997), 119–139. https://doi.org/10.1006/jcss.1997.1504

[13] J. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *Annals of Statistics* 29 (2001), 1189–1232.

[14] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate String Joins in a Database (Almost) for Free. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, Roma, Italy, 491–500. http://www.vldb.org/conf/2001/P491.pdf

[15] Luis Gravano, Panagiotis G. Ipeirotis, Nick Koudas, and Divesh Srivastava. 2003. Text Joins in an RDBMS for Web Data Integration. In *Proceedings of the 12th International Conference on World Wide Web* (Budapest, Hungary) *(WWW '03)*. Association for Computing Machinery, New York, NY, USA, 90–101. https://doi.org/10.1145/775152.775166

[16] 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 (Aug. 2019), 1986–1989. https://doi.org/10.14778/3352063.3352116

[17] Peter Norvig. 2009. Natural language corpus data. In *Beautiful data*. O'Reilly Media, Boston, USA, Chapter 14, 219–242.

[18] 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. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[19] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. 2009. A machine learning approach to foreign key discovery. In *WebDB*. Providence, Rhode Island.

[20] A Shlosser. 1981. On estimation of the size of the dictionary of a long text on the basis of a sample. *Engineering Cybernetics* 19, 1 (1981), 97–102.

[21] Jiannan Wang, Guoliang Li, and Jianhua Fe. 2011. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, IEEE, Hannover, Germany, 458–469.

[22] Egon L Willighagen, Andra Waagmeester, Ola Spjuth, Peter Ansell, Antony J Williams, Valery Tkachenko, Janna Hastings, Bin Chen, and David J Wild. 2013. The ChEMBL database as linked open data. *Journal of cheminformatics* 5, 1 (2013), 1–12.

[23] Yi Zhang and Zachary G. Ives. 2019. Juneau: Data Lake Management for Jupyter. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1902–1905. https://doi.org/10.14778/3352063.3352095

[24] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: Parallel Databases Meet MapReduce. *The VLDB Journal* 21, 5 (Oct. 2012), 611–636. https://doi.org/10.1007/s00778-012-0280-z

[25] 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* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 847–864. https://doi.org/10.1145/3299869.3300065

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