Discovering Related Data At Scale [Scalable Data Science]

Sagar Bharadwaj Praveen Gupta Ranjita Bhagwan Saikat Guha
Microsoft Research
{t-sabhar, t-pravgu, bhagwan, 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 automatically learn related sources of data. Leveraging queries made to the data lake and several lightweight features, we build a classifier that says whether two columns across two data streams are related. Our model has high accuracy at 96% and even with metadata-only features, we achieve an accuracy of 95%. We build a graph of related columns on a dataset of size over 4 Petabytes in around 80 minutes, showing that our techniques scale very well to large datasets. We also show that our model discovers several interesting relations not seen before in queries, thereby demonstrating the value of our approach.

PVLDB Reference Format:
. . *PVLDB*, (): xxxx-yyyy, .
DOI:

1. INTRODUCTION

Data analysis tasks frequently need multiple data streams [1] that collect information about various entities owned and operated by different organizational groups. 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 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 key relationships often capture related data, in large unstructured data lakes, data often sits in isolated organizational silos with no obvious ways to relate data streams to each other.

To address this problem, we believe the right approach is to build a "data graph" that captures related data sources. Such a graph can have many applications. For instance, given a task and an initial set of data streams, an analyst can query the graph to determine all related data streams. Building such a graph boils down to answering the question, "Given a column in a data stream, what are the related columns in other data streams to it in the data lake?". Previous work has addressed this question [23, 8], but these techniques do not scale well to exabyte-scale data lakes.

While the immense scale of data lakes poses a big 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 have built machine-learning models that automatically discover related data sources. 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 several metadata-specific features in our model. We also use data-based features inspired by previous-work [23, 8] derived from sampling the top 1000 rows of the column since scanning all data and even randomly sampling data from columns are prohibitively expensive.

In this paper, we evaluate two models: a *metadata-only* model, which learns characteristics of related columns using only metadata-based features, and a *complete* model which uses both metadata and data-based features. Our results, quite counter-intuitively, show that the metadata-only model which uses only 2 features derived from metadata is able to detect related columns with 95% accuracy. Adding data-based features does improve accuracy but by a marginal 1%.

Our paper makes the following novel contributions:

- We propose the use of machine-learning on queries made to data streams, specifically `JOIN` clauses, to learn characteristics of related data columns.

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

Proceedings of the VLDB Endowment, Vol. , No.
ISSN 2150-8097.
DOI:

- We propose the use of two metadata-based features: *embedding-enhanced column-name similarity* and *column-name uniqueness*. We use data-based features as well, but instead of calculating them on full data, we base them on samples of the top 1000 rows. This is to ensure that our techniques scale well.

- We evaluate the models using data of one large service, namely Office365 Core. Additionally, we build a data graph using this model for one of Microsoft's internal services and evaluate how well it detects useful relations not seen before in any queries.

Our experiments with Office365 Core data show that the metadata-only model does very well with a precision of 96% and recall of 93% on test data. By adding data-based features these numbers increase to 97% and 96% respectively. Our evaluation of the internal service's data graph shows that the complete model detects 477,000 new relations (i.e. not seen before in our query set) with a precision of 97%.

2. BACKGROUND

In this section, we first provide a brief overview of Cosmos, Microsoft's big-data system, and Scope, the language for processing data on Cosmos. For a more in-depth description of Cosmos and Scope refer to [3, 22]. Next, we provide examples on how analysts use Cosmos and Scope for various tasks, motivating the need for automated data discovery.

2.1 Cosmos and Scope

Cosmos [3] stores multiple exabytes of mostly telemetry data and is used daily for analytical jobs by every Microsoft product team. Data in Cosmos is stored in either unstructured or structured streams. Unstructured streams, similar to files in a filesystem, 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 filesystems and databases, however, 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 above constraints result in a small number of highly optimized best-practice workflows. Data, such as webserver logs, is initially ingested as unstructured streams (raw). A processing script then cleans and transforms the raw data into a structured stream (intermediate). Another script then enriches the intermediate data 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 typically structured to enable efficient querying. The raw and intermediate streams are not meant for consumption by other analysts; hence, the rest of this paper focuses only on these cooked structured streams.

Data is processed and consumed using Scope [22] 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 result 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 3.3 we describe how we parse this generated query plan to create ground-truth for our models.

Teams often provide multiple *Views* of their cooked streams. A View is a concept in Scope, similar to a (non-materialized) SQL view, where a (logical) dataset is generated at query time as some transformations over an underlying (physical) dataset. This is primarily done for forward compatibility of downstream consumers to breaking schema changes in the underlying cooked stream. A secondary reason is to include computed columns or joins that are too expensive to store separately in the cooked stream. 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.

Scale: Cosmos stores multiple exabytes of data in many 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.

2.2 Examples

We outline three analytics scenarios of how analysts are using data in Cosmos. While they are not entirely representative of the million tasks that run on Cosmos every day, they illustrate the need for an automated data discovery solution that goes beyond the state-of-the-art.

2.2.1 Network Usage Attribution

Large services make heavy use of wide-area networks which are expensive to maintain. The purpose of this analysis was to attribute network usage to specific components in the service so that engineers could optimize components that use considerable amounts of network bandwidth, thereby reducing costs. The analyst had to join three sources of data: a) activity of each component of the service, i.e. number of requests made to it indexed by time, b) a mapping of components to servers (and hence data-centers) indexed by time and c) Bandwidth usage, both incoming and outgoing, per data-center, indexed by time. While the first and second data source were available within the same service's data lake, the individual teams generating the data were different. To discover these data streams, the analyst had to manually approach various engineers for domain-knowledge, a process that took her many days. A separate networking service owned the bandwidth usage streams, and discovering it took the analyst almost three weeks.

2.2.2 Collaboration Graph

A collaboration graph captures individuals who collaborate or work together. To build this using data, the analyst had to collect logs from various enterprise services: a) email logs link senders to the recipients of the email, b) chat logs also provide clusters of individuals who communicate frequently with each other, and c) calendar logs provide information about individuals who attend the same meetings. In this case, the analyst did have access to a catalog that implemented simple search on column names. However, he had to guess appropriate search keywords to find these logs

and though better than manually sifting through various streams, the process was still laborious.

2.2.3 Root-cause Analysis in Services

Orca [2] is a tool that root-causes issues in service deployments to the appropriate buggy code-commit. To build Orca, the analysts needed access to a) server-specific deployment information, b) a mapping from server to the version of the software running on that server, and c) a mapping from software version to commit information. Again, getting all this information from different teams and organizations was challenging and took many weeks.

3. APPROACH

In this section, we first provide motivation for why we concentrate on building models that use metadata-based features and data-based features built upon sampling only the top k rows of columns. Next, we provide an overview of our approach. Finally, we describe each component in our design in detail.

3.1 Scaling Challenges

Metadata access is significantly faster than accessing data. In Cosmos, 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 any approaches that can work in metadata-only mode are going to be significantly faster today, and more so in the future.

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. Our data access requirements (for non metadata-only approaches), however, are dramatically different. Specifically, we require a tiny number of rows, say 1000 rows (less than 0.5%), but for a large number of files. Ideally these rows would be sampled at random, however due to large extents and lack of random access within extents, random sampling essentially requires reading all extents from disk, which is extremely resource intensive. Sequentially accessing top rows from the first few extents is faster, however, the sampled data is skewed due to clustering and sort order of the stream. We do not envision Cosmos being optimized for our workload. We thus pick a design point that trades off sampling uniformity against performance, i.e. we use metadata and sample only the top 1000 rows. Section 3.4 further quantifies the costs that drove this decision.

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. Uniformly sampling data from a View, however, requires materializing the entire view. Even retrieving the top 1 row from a view may involve a resource intensive computation (e.g. aggregation) before any row is produced.

Lastly, 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.

3.2 Overview

Our objective is to build a data graph that connects related columns. We have designed a machine-learning approach and build both a metadata-only model to cater to scenarios where we have no access to data, and a complete model that combines meta-data based and data-based features from our sample. We then use these models to build the data graph. We outline four main steps towards learning related columns and evaluating the model:

Parsing queries: For each Scope job, the compiler generates a query plan. Our query plan parser discovers `JOIN` clauses, each of which yields at least one pair of related columns. We label these pairs as "positive" samples for learning the classifier. To determine "negative" samples, we randomly choose pairs of columns from randomly chosen stream pairs which we have not seen occurring together in any of the parsed `JOIN` clauses. Section 3.3 provides more details on the parser.

Extracting Features: For every sample in our data set, we extract *metadata based features* that we get from Cosmos's metadata store and *data based features* calculated over the top 1000 rows of a column. Section 3.4 describes the features that we use.

Machine learning: Using the positive and negative pairs and their features, we then use a random-forest classifier that learns whether two columns are related or not. We learn three classifiers: a *metadata-only* classifier which uses only metadata-based features, a *data-only* classifier that uses only data-based features, and a *complete* classifier that uses both metadata-based and data-based features. We use the data-only classifier only for evaluation and comparison. Section 3.5 describes the learning process in detail and Section 4.1 describes our evaluation and comparison of these models.

Discovering related columns: Finally, we use this classifier to find related pairs of columns to construct the data graph. Since passing all possible column-pairs in a data lake to the model is infeasible, we prune this set using two techniques: using a reverse-index on column contents, and using metadata clustering. Section 3.6 describes our pruning approach while 4.2 describes how scalable our data graph creation is and evaluates its precision.

3.3 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

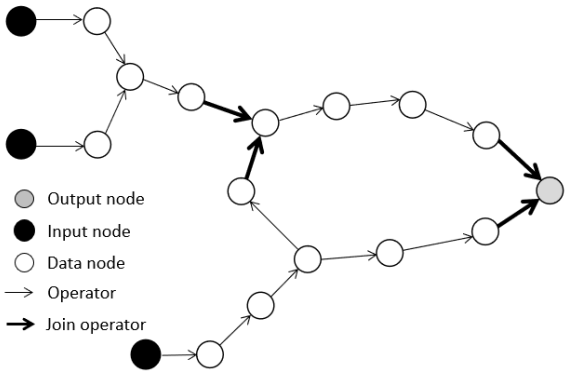

Figure 1: query plan of a Scope Script.

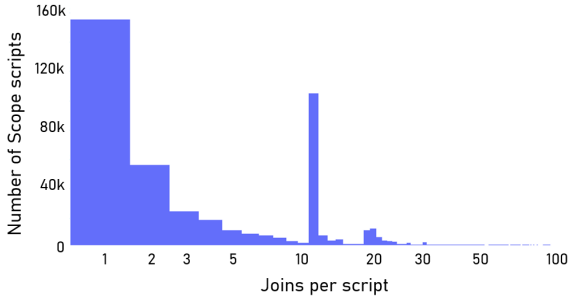

Figure 2: Histogram of Join statements per script

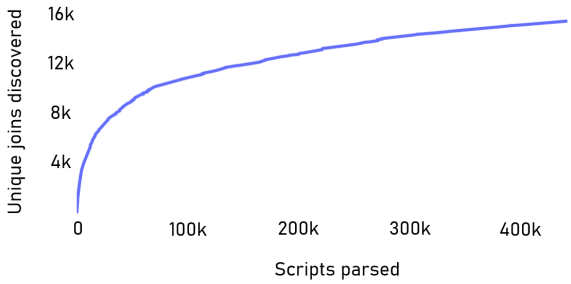

Figure 3: Fraction of unique join pairs discovered against fraction of scripts parsed.

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. Since the Scope compiler does not understand UDFs, 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 [9] that help us capture such complex semantics in the column names of such pairs (details in Section 3.4).

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 small substring of 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, 11, 12], fuzzy string matching [20, 5] etc we do not resort to these methods as column metadata captures such relationships in most cases 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.

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 joining data that 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 3.4.

3.4 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 difference between sampling-based feature extraction, and metadata-based feature extraction.

3.4.1 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 [16] to further subdivide the tokens into recognizable English words

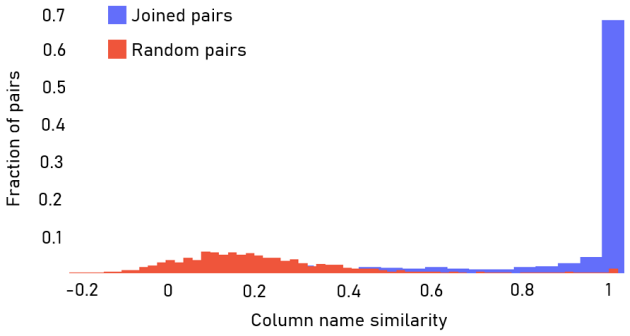

Figure 4: Distribution of column name similarity score.

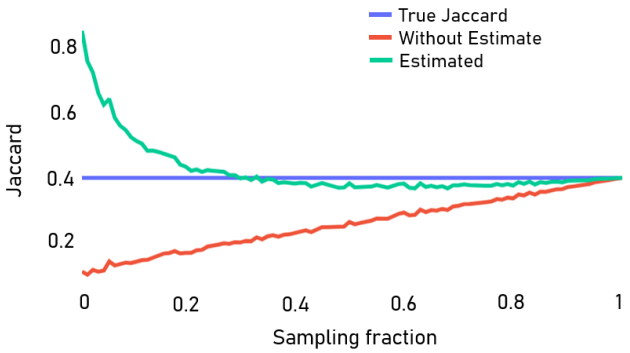

Figure 5: Behaviour of Jaccard similarity estimate for a pair of columns with a highly skewed distribution.

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: 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 [9] 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 it's most unique token. Given a column name, it's uniqueness score is same as the maximum ITF among the ITF of its individual tokens. We use the uniqueness scores of both column names as two separate features.

3.4.2 *Data-based Features*

For each column-pair we sample the top 1000 entries of both columns. Here, we make the simplifying assumption that our sample is approximating a random sample from the column, and calculate the following:

Number of Unique Values: Estimating number of distinct values from samples has been well studied in literature [13, 19, 4, 14]. However, negative results prove that it is impossible to get good error bounds with high probability in the distinct value estimation problem, without a large sample, for worst case distributions [4]. In practice however, worst case distributions are not frequent and such estimates can be used [14]. In our experiments we use Shlosser's estimate [19] which is fast to compute and hence enables quick calculations on millions of pairs of columns. We use the distinct value estimates of the two columns in the pair under consideration as two separate features.

Jaccard Similarity: 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. 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 [19]. 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 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

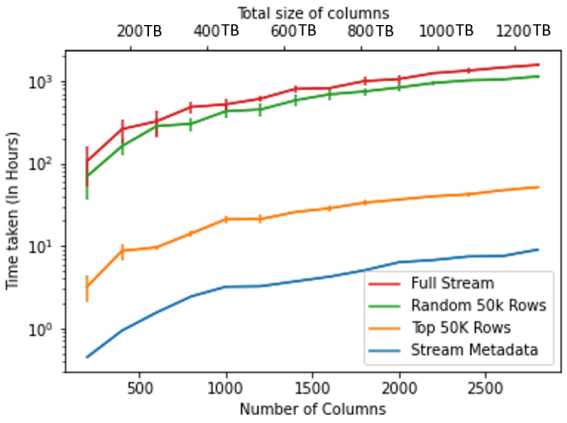

Figure 6: Performance comparision for scanning full column, randomly sampling, top sampling and crawling for metadata.

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 relatedness and can be used as a feature for the machine learning model.

Inclusion Dependence: Some column pairs may have a low Jaccard similarity score 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 numbers 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.

3.4.3 Performance of Feature Extraction

Figure 6 explains the performance difference between scanning full columns, randomly sampling from the columns, taking the top 50k values from columns as a sample, 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 3.1 has provided more 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 sampling the top rows.

3.5 Machine Learning

We use a random forest-based classifier since it gives the best results on our experimental data. Negative sampling is based on non-existence in the set of `JOIN` clauses. These can have some false negatives, but we assume that such incorrectly labelled data will be small in number. We use a random forest with 100 trees and the predicted probability is the weighted average of probability estimates of individual trees. The higher the probability, the more related a column-pair. We can use any cut-off probability to determine if a column-pair is related. In our evaluation in Section 4.1 we use a cut-off of 0.5.

We learn two models: a metadata-only model which uses column name similarity and column name uniqueness scores of the two columns as features and a complete model that uses all features described in Section 3.4. For the sake of comparison and evaluation, we build a data-only model as well which uses only the data-based features.

3.6 Pruning and Discovery

Finally, we build the data graph using the model using two steps. In the *pruning* step, we find candidate column-pairs to input to the model, and in the *discovery* step, we use the model to discover related column-pairs. Since the number of columns in a data lake is rather large we have to prune the set of column-pairs to input to the model. In keeping with our approach of treating metadata and data separately, we adopt a two-pronged approach to pruning column-pairs, using a union of metadata-based pruning based on k-means clustering and data-based pruning based on creating a reverse-index of sample data.

Reverse index-based Pruning. There is extensive literature on content-based pruning using top k set similarity search [23, 6, 10, 1]. 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. However, we can 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 column 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.

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

Metric	metadata-only	data-only	complete
Accuracy	0.95	0.87	0.96
Precision	0.96	0.86	0.97
Recall	0.93	0.87	0.96
F1-Score	0.94	0.87	0.96

Table 1: Precision, Recall and F1-Score of metadata-only model, data-only model, and complete model.

Metric	equality	+embedding	+uniqueness
Accuracy	0.82	0.89	0.95
Precision	0.99	0.88	0.96
Recall	0.65	0.91	0.93
F1-Score	0.78	0.89	0.94

Table 2: Precision, Recall and F1-Score increases with each added feature in the metadata-only model.

cluster form a candidate column-pair for the data graph. We eliminate any pair which span two different clusters.

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. Clustering on column name embeddings minimizes the possibility of such eliminations from potential candidates list. We input these candidates to the model and finally get a list of related column-pairs, or edges in our data graph.

4. EVALUATION

We first perform a conventional cross-validation based evaluation of the models by dividing the query data into train and test sets. Next, we evaluate the scalability of our approach to create the data graph. Finally, we perform a manual labelling experiment to see how well the data graph captures new relationships which have not been seen before.

4.1 Model Comparison

This evaluation is done in Python using the sklearn toolkit. From the Office365 Core dataset, we use 4045 positive examples or `JOIN` pairs and 4045 negative examples or randomly selected column-pairs to learn the classifier. We evaluated ten different classifiers and found the Random Forest classifier with 100 estimators provides the best results.

Table 1 shows the accuracy, precision, recall and F1-score for the metadata-only, data-only, and the models respectively. The metadata-only model performs surprisingly well with an accuracy of 95%, while adding the data-based features in the complete model improves the metrics (accuracy = 96%), though marginally. The data-only model does not perform as well as the metadata-only model; it provides an accuracy of only 87% 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 3. The table shows that 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 (0.99). When we add column name similarity, the recall shoots up to 0.91, but the precision drops from 0.99 to 0.88. Adding column name uniqueness brings the recall 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 losing out on interesting relationships between columns that don't 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 , and why they both add value.

TP in metadata-only, FN in data-only. When related column-pairs contain randomly generated IDs that refer to the same entitie such as users, the columns have a very high cardinality, as a result of which the intersection size of their samples may be 0. In such examples, content based features such as Jaccard similarity score 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` are likely related and can be captured by embedding-enhanced column name similarity.

TP in data-only, FN in metadata-only. In some cases, users use shortened column names which cannot be captured by the meta data features. For example, two columns named `ManagerName` and `mngr` may be related, but only the data based features will be able to capture the similarity.

TN in metadata-only, FP in data-only. Integer columns that record counts of an occurrence of an event may seem related to another column that is just an integer ID. For example two columns named `fileID` and `finished_count`, both of which have the same sequence of integers, may be flagged as being related if the column names are not considered.

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 different version control systems. One follows a GUID format, while the other follows the integer format. Hence they refer to different entities and are not related.

4.2 Pruning and Discovery

In this section, we first evaluate the performance of the reverse index-based and metadata-based pruning techniques. Next, we evaluate how our models do in practice by inputting the candidate column-pairs to the metadata-only and the complete models.

Evaluation Dataset: We have learned our model using data from Office365 Core's data lake. Hence, to evaluate its efficacy at discovering new relations, we use data from the service that supports DevOps within Microsoft. This data lake has 137,000 streams and a total of 2.6 million columns.

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

Table 3: End-to-end performance of our implementation.

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. The total size these data streams is 4.5 Petabytes.

30,754 columns can create in excess of 900 million column-pairs. Our pruning reduced this to approximately 40 million pairs. By inputting these pairs to the model, we built a data graph with 12 million edges that had a probability > 0.5, of which 2 million edges had probability > 0.9.

Scalability: Table 3 outlines the performance of our implementation that creates the data graph, starting with metadata collection and data sampling, all the way until model prediction. All steps are implemented in Scope and run on Cosmos. To achieve scale and reduce latency, we have parallelized several of the steps in this process. 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.

Quality: To evaluate the quality of the data graph, we emulate a simple application on the graph. We choose a random set of 10 columns and for each column, we query the data graph and get the top ten related columns sorted by the probability that the model outputs (most related to least related). Two of the authors of this paper independently examined these column-pairs, the streams, and their context, and manually labeled each relation as a true-positive or a false-positive. To combine their labels, we say a column-pair is a true-positive if both labelled it as a true-positive. Else, we label it as a false-positive. Of the 100 column-pairs labeled, 97 were true-positives and only 3 were false-positives, thereby yielding a high precision of 97%.

Here are some interesting examples of columns that we found were related which were not present in any form 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.

5. LIMITATIONS AND FUTURE-WORK

While we have made significant progress on the problem, there are several ways to further improve accuracy, precision and recall. For instance, to create our ground-truth, we could consider `UNION` statements in addition to `JOIN`. Also, currently we consider only single column-level similarity, though similarity could be context-sensitive as well. For instance, a column named `Id` in a stream that also has a column named `UserName` could actually be a "User Id". Such expansion would further help the metadata-only model. We intend to build such context-sensitive models in the future.

Previous work has also looked at discovering related data from slightly dissimilar data [5, 20, 11, 12]. This may be useful in cases where transformations are used before the `JOIN` clause. We intend to augment our techniques using these as well.

We derive our ground truth by parsing query plans. As we describe in Section 3.3 this has some limitations. We can improve our algorithms by leveraging the data flow graphs and understanding provenance of streams better [18]. We leave this for future-work.

6. RELATED WORK

Data Civilizer [8] discovers related data in a two-step process. It first profiles available datasets to extract datatype, cardinality and a column signature (distribution for numerical values and inverted word frequency vector for textual values) to build a linkage graph with lightweight relationships like column 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 [17]. Although it allows users to search for relationships between data nodes using several similarity measures as mentioned above, it needs a full pass on candidate data columns for calculating the heavy relationships.

JOSIE [23] is a top K overlap set similarity algorithm for finding joinable columns. It uses an inverted dictionary which maps every distinct token in a datalake 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 column, 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 potentially be in numerical formats.

Several efforts have described various data-based similarity detection techniques [24, 15, 7, 21] for finding related tables in a corpus of heterogeneous tables. However, these systems do not specifically address the issues we face in our exabyte-scale data lake. They require full scans of data which, as we have shown, is not feasible at scale.

7. 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 which analysts may otherwise take a long time to discover.

8. REFERENCES

[1] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*, pages 131–140, 2007.

[2] R. Bhagwan, R. Kumar, C. Maddila, and A. A. Philip. Orca: Differential bug localization in large-scale services. In *13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, October 2018. Won the Jay Lepreau Best Paper Award.

[3] R. Chaiken, B. Jenkins, P.-r. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[4] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 268–279, 2000.

[5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 313–324, 2003.

[6] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 5–5. IEEE, 2006.

[7] P. H. Chia, D. Desfontaines, I. M. Perera, D. Simmons-Marengo, C. Li, W. Day, Q. Wang, and M. Guevara. Khyperloglog: Estimating reidentifiability and joinability of large data at scale. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 350–364, 2019.

[8] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. 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, 2017.

[9] V. Efstathiou, C. Chatzilenas, and D. Spinellis. Word embeddings for the software engineering domain. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 38–41, New York, NY, USA, 2018. Association for Computing Machinery.

[10] F. Fier, N. Augsten, P. Bouros, U. Leser, and J.-C. Freytag. Set similarity joins on mapreduce: An experimental survey. *Proceedings of the VLDB Endowment*, 11(10):1110–1122, 2018.

[11] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, et al. Approximate string joins in a database (almost) for free. In *VLDB*, volume 1, pages 491–500, 2001.

[12] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *Proceedings of the 12th international conference on World Wide Web*, pages 90–101, 2003.

[13] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB*, volume 95, pages 311–322, 1995.

[14] R. Motwani and S. Vassilvitskii. Distinct values estimators for power law distributions. In *2006 Proceedings of the Third Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 230–237. SIAM, 2006.

[15] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena. Data lake management: Challenges and opportunities. *Proc. VLDB Endow.*, 12(12):1986–1989, Aug. 2019.

[16] P. Norvig. Natural language corpus data. *Beautiful data*, pages 219–242, 2009.

[17] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *WebDB*, 2009.

[18] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing. Bootstrapping privacy compliance in big data systems. In *2014 IEEE Symposium on Security and Privacy*, pages 327–342, 2014.

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

[20] J. Wang, G. Li, and J. Fe. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *2011 IEEE 27th International Conference on Data Engineering*, pages 458–469. IEEE, 2011.

[21] Y. Zhang and Z. G. Ives. Juneau: Data lake management for jupyter. *Proc. VLDB Endow.*, 12(12):1902–1905, Aug. 2019.

[22] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. Scope: Parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, Oct. 2012.

[23] E. Zhu, D. Deng, F. Nargesian, and R. J. Miller. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 847–864, New York, NY, USA, 2019. Association for Computing Machinery.

[24] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller. Lsh ensemble: Internet-scale domain search. *Proc. VLDB Endow.*, 9(12):1185–1196, Aug. 2016.