Extracting Syntactic Patterns from Databases

Andrew Ilyas, Joana M. F. da Trindade, Raul Castro Fernandez, Samuel Madden

CSAIL, MIT <aiilyas,jmf,raulcf,madden>@csail.mit.edu

***Abstract*—Many database columns contain string or numerical data that conforms to a pattern, such as phone numbers, dates, addresses, product identifiers, and employee ids. These patterns are useful in a number of data processing applications, including understanding what a specific field represents when field names are ambiguous, identifying outlier values, and finding similar fields across data sets.**

One way to express such patterns would be to learn regular expressions for each field in the database. Unfortunately, existing techniques on regular expression learning are slow, taking hundreds of seconds for columns of just a few thousand values. In contrast, we develop XSYSTEM, an efficient method to learn patterns over database columns in significantly less time.

We show that these patterns can not only be built quickly, but are expressive enough to capture a number of key applications, including detecting outliers, measuring column similarity, and assigning semantic labels to columns (based on a library of regular expressions). We evaluate these applications with datasets that range from chemical databases (based on a collaboration with a pharmaceutical company), our university data warehouse, and open data from MassData.gov.

I. INTRODUCTION

Modern enterprises store their data in a wide range of different systems, including transactional DBMSs, data warehouses, data lakes, spreadsheets, and flat files. Data analysts often need to combine data from these diverse data sets, frequently incorporating external data from even more sources. A key challenge in this setting is finding related data sets that can be combined to answer some question of interest.

As an example, analysts at Merck—a pharmaceutical company—often need to join tables that contain chemical compounds. Unfortunately, there are at least three identifier formats (e.g., InChI, InChIKey, and SMILES, shown in Fig. 1) used internally in Merck, not to mention additional formats that may be used in external data sources. Because of this diversity of ID formats, a simple text search is not sufficient to find relevant tables—attribute names are different. Indeed, they cannot even perform an approximate search to find similar content as these identifiers are not comparable. Manually building a mapping between the identifiers in the different formats and creating a lookup table is an expensive option.

Inchi	InchiKey	SMILES
InChI=1S/C2…	UHTHHESE…	COc1cc2c(Nc…
InChI=1S/C3…	UZLMEAPB…	O[C@@H]1[C@…
InChI=1S/C2…	HYNYUFZP…	O[C@@H]1[C@…

Fig. 1: Example of different chemical compound ID formats

A better option would be to label the relevant attributes with useful metadata, e.g., assign a *chemical identifier* label to all identifier columns in the table that represent. Unfortunately, manual labeling is also infeasible in an company with large volumes of data: it requires a great deal of time and is error prone as it may miss many tables that contain relevant information, especially when considering external data.

To address this problem, we observe that many relevant attributes in enterprise databases are *highly structured*, i.e., they follow simple syntactical patterns. For example, in Fig. 1 the InChi number always starts with the pattern *InChI=* and the InChiKey is a 14-character followed by a hyphen, followed by a 10-character followed by another hyphen and an additional character [1]. More common examples of structured attributes are dates, product identifiers, phone numbers, enumerated types (gender, etc.), and so on. Often these columns are stored as strings in the database, but if they could be labeled with richer structural information about the format of values, indexing, searching and comparing values, and finding exceptional or outliers values, could be done much more efficiently.

In this paper, we introduce **XSYSTEM**, a method to extract syntactic patterns from datasets into data structures called XTRUCTUREs. A XTRUCTURE represents syntactic patterns, and can be compared with other XTRUCTUREs as well as regexes. Once XSYSTEM learns a collection of patterns, analysts can use them to conduct several commonly performed tasks, including: *automatic label assignment*, where data items are assigned a class by comparing them to a library of known classes (written as regexs or XTRUCTUREs); *finding syntactically similar content*, where learned XTRUCTUREs are compared to see if they are similar, and *outlier detection*, where a learned XTRUCTURE for a single item is compared to other XTRUCTUREs to check that its structure is different. These applications share two common requirements: (i) XTRUCTUREs must be quickly synthesizable and (ii) XTRUCTUREs must be comparable to each other and to regular expressions.

In addition to supporting these requirements, XSYSTEM must: i) be able to work without human intervention, as neither semi-automatic nor interactive tools scale for large amounts of data; ii) learn syntactic patterns fast, which calls for both an asymptotically efficient model as well as a parallelizable implementation; and iii) be quickly synthesizable and manipulatable given only raw datasets, since this is all that many analysts may be able to initially access. Speed of learning is crucial for real world scenarios, as not all data analysis tasks can cope with stale data.

XSYSTEM. Our approach learns syntax from examples incrementally. For each example, it exploits the existence of delimiters in known entities to split the problem of extracting the pattern into learning the syntax of each of the *tokens* separated by those delimiters. The underlying data structure

used to learn each token is a branching linear distribution sequence that is equivalent to a Deterministic Acyclic Finite State Automaton (DAFSA), which is asymptotically simpler to learn than minimal Deterministic Finite Automata (DFA), often used in regular expression learning. The learning procedure relies on a *branch and merge* strategy that allows us to incrementally adapt a prior to new observed examples. This permits us to capture different syntactical structures that appear in the same column. This branch and merge strategy is also at the center of the parallelization approach used in XSYSTEM.

We evaluate XSYSTEM on the three applications mentioned above on real datasets ranging from our university's data warehouse, open government data and a public chemical database. We find that XSYSTEM can form a syntactic representation of given data much faster than automatic DFA learners, and that we can use it effectively for our target applications.

II. Related Work

In this section we discuss our contributions in the context of several techniques and research areas related to XSYSTEM.

Information Extraction. XSYSTEM is related to information extraction (IE) in that it extracts a structural representation from data. Most IE techniques extract structure from totally unstructured data, such as text, or semi-structured data, such as XML and HTML. In addition, most of those techniques require variable amount of human input. XSYSTEM must work automatically and it operates on structured data, producing one succinct pattern that represents the syntactic structured of a collection of input strings. XSYSTEM complements the existing techniques in IE and achieves good performance in important applications to large enterprises.

Regular expression inference. These techniques extract syntactical patterns from collections of strings. The most recent work uses multi-objective optimization and aggressive space pruning to reduce the running time [2] of the inference process. Performance is still an issue for the method to be used in enterprise settings as their evaluation shows—more than 40 min for learning a dataset with 500 entries with 32 threads. XSYSTEM reduces the unnecessary expressiveness of regular expressions to gain in performance, as we will justify next.

Other methods can be divided into whether negative examples are required or not. Those that require negative examples are rarely suitable in enterprise settings. Out of systems that only require positive examples, [3], [4], and [5] are the most relevant. With [3] we share our treatment of input characters as their character class (referred to as token class in their case) to produce a higher level abstraction of input data. Their method learns a cyclic DFA, while we will show in this paper this expressiveness is not necessary for the applications we target. In the context of XML, the method in [6] learns concise regex from a few positive examples, and it is also possible to generate readable strings from the representation the method learns, opening an avenue for comparison. Lastly, ReLIE [4] requires example regular expressions, that are then further refined. We differ in that we operate without human input. Unlike all this work, we focus on: i) designing XSYSTEM to capture syntactical patterns in databases, and not to solve the general–and more complex–problem of learning regular expressions for infinite languages; ii) support efficient comparison of the learned patterns, which we have shown helps in identifying syntactically similar content, automatically labelling data, and identifying syntactic outliers.

Program Synthesis. Program synthesis based methods have seen a surge in popularity [7]–[9]. Unlike XSYSTEM, their goal is typically to operate and transform data, for example for data cleaning. This means that the complexity of the structured they need to build and maintain internally is higher than that of XSYSTEM. For example, BlinkFill [10] must build an *InputDataGraph* to then transform the data that is more expensive to build than XSYSTEM, and unnecessary for our goal. Other techniques, such as [7], [8] require negative examples and differ again from our automatic technique.

III. Motivation and Requirements

In this section, we use three applications that motivate XSYSTEM and its requirements.

1. Automatic Label Assignment.

Automatic label assignment attaches a semantic type (e.g. "chemical compound ID", "phone number") to columns in a dataset, so that users can understand the content of columns and perform semantic search for similar types of columns. A key observation is that many semantic types are already available in regex libraries [11], and for important semantic types inside an organization, writing such a regex is relatively straightforward. For example, for the examples of Fig. 1.

XTRUCTUREs can be used only to determine *syntactic* similarity and *not semantic* similarity. However, given a table of (regex, semantic label) pairs, it is possible to learn a XTRUCTURE for each attribute in the database, and then perform a search for *syntactically* similar regexes. When there is a match, the label associated to the regex is associated to the column represented by the XTRUCTURE. This introduces two key requirements for XTRUCTUREs: they must be 1) fast to learn, since we need to infer them for every column in the database, and 2) comparable to regular expressions.

2. Summarization and Attribute Comparison. Once some interesting attributes are identified, data analysts often wish to find other similar attributes across datasets (e.g., to obtain candidates for joining two datasets together.) Rather than comparing each pair of attributes using set-similarity joins or approximate methods [12], [13], we can learn a more compact representation, e.g., a XTRUCTURE for each attribute in a database. We can then compare these representations instead of the raw data, which offers the additional benefits of human interpretability and reduced I/O usage.

Last, the cost of learning the XTRUCTURE is paid only once, and can be reused subsequently for other applications as we are describing in this section.

To be useful for summarization and comparison, XSYSTEM must learn human-readable XTRUCTUREs, similar to regexes in common programing languages, and XTRUCTUREs must be comparable to one another, to permit finding similar content.

CHEMBL	Date	ZIP Code
CHEMBL102034	12/27/2016	2111
CHEMBL102036	12/12/2015	2127
CHEMBL102037	01/24/2003	2110
CHEMBL102038	02/15/87	121091402
CHEMBL102040	10/10/86	2215

Fig. 2: Examples attributes

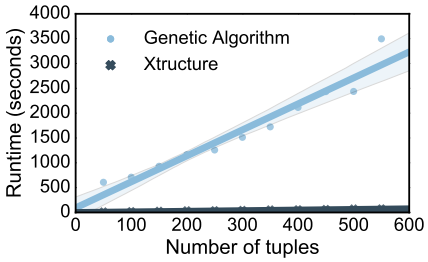

Fig. 3: XSYSTEM vs. regex learning

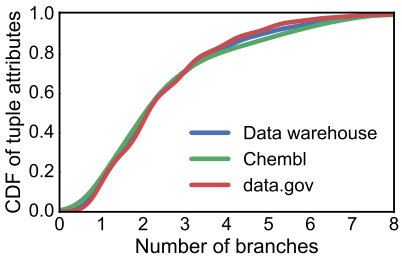

Fig. 4: Branches per tuple attribute.

3. Syntax-Based Outlier Detection. One long-standing problem in data management is concerned with data quality; in particular, errors occur frequently, whether due to data entry or anomalous values or readings [14].

We observe that by learning the syntactic pattern of an attribute, we can detect many types of errors, particularly those that are *syntactic outliers*, i.e., elements that do not closely the match a learned XTRUCTURE. Consider the example of Fig. 2, with real ZIP codes from Boston. The 4th cell value an erroneous ZIP code. In this case, it is possible to detect that it has a different length than other records in the same attribute and does not fit the general syntactic pattern of the column.

To be able to detect syntax-based outliers, XTRUCTUREs must support the concept of a *scoring fit*, i.e., a numeric score capturing how well a value fits a learned XTRUCTURE. Also, XSYSTEM to be used as an outlier detector, it must not overfit the XTRUCTURE to all the values, or it will not detect outliers. Instead, it must represent the general syntactical pattern and not capture the content of a few outliers.

A. Motivation for a New Approach

Although the above requirements could be satisfied by learning regular expressions (DFAs), regular expression learning [4], [5], [15] is an NP-complete problem, and what that means in practice is that solutions are extremely inefficient.

How inefficient is to learn regex?. To build intuition about this inefficiency, we used a state-of-the-art regex inference algorithm [2] to learn a regex over a few hundred tuples and found that it took around an hour to complete. Figure 3 shows the speed of learning a XTRUCTURE from data using XSYSTEM with the state of the art algorithm [2]. Here we show the time to learn a regular expression or a XTRUCTURE over a column, as the length of the column (in tuples) grows. The genetic algorithm based method is infeasible for our target applications because it takes thousands of seconds to learn a regular expression for a single column, making it impractical to use in even a moderate collection of databases with a few hundred columns. In contrast, the performance of XSYSTEM with XTRUCTUREs grows sub-linearly with the number of tuples, as we will show in subsequent sections.

If regular expressions were available, we could use them to solve the application scenarios we showed above. However, because regular expression learning algorithms solve a more complex problem than what is needed for the applications we have identified at a high computational cost, we sought a simpler language that is both efficient to learn and that is sufficient to capture the structure of many database columns.

The Opportunity. Fortunately, we have observed that real data in databases is often quite simple, and does not require the full expressivity of DFAs/regular expressions. In particular, most attributes in database have the following properties:

- **Simple structure.** Through the wildcard "*" and "+" operators, regexes allow infinite variability of structure within a domain. In practice, on the MassData dataset (open data from Massachusetts), we found that around 20% of columns are fixed length, over half have only 3 distinct column lengths, more than 85% have average length less than 10, and 99% have average length less than 50, and similar trends are also present in ChEMBL and data.gov. This makes sense because databases are designed to be easy to manipulate and process; constraining the data formats into well-structured values helps achieve this goal. Further, many regular expression learning papers focus on learning a *minimal* regular expressions, but since database columns are already simple, minimality is not a primary concern, especially if it comes at the cost of efficiency.
- **Consistent structure.** The optionality operator in regular expressions allows one to construct concise expressions such as "AB(C)D." Instead, the equivalent "*ABD|ABCD*", which separates each pattern into a different *branch* is simpler to learn. Fig. 4 shows that 40% of the attributes of data.gov and two other datasets are representable by 2 global branches, and nearly 100% by 8 branches. Again, regular expressions unnecessarily favor expressivity over efficiency.

In short, regexes are neither necessary (too expressive) nor sufficient (they are too slow) for solving the problem of structure learning addressed in this paper. Instead, as we show, less complex XTRUCTUREs can be learned more efficiently while still capturing the structure of real databases.

IV. XSYSTEM IMPLEMENTATION

In this section we introduce the XTRUCTURE model to learn syntactical patterns from structured data.

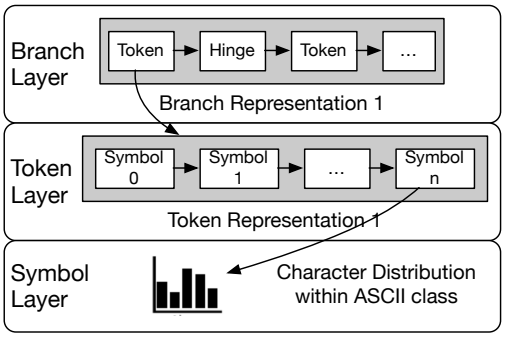

Fig. 5: Xtructure data model

A. The XTRUCTURE Model

The goal of XSYSTEM is to learn a XTRUCTURE from examples $\{T\} \subseteq A$ incrementally (tuple by tuple). To do this, we design an architecture that allows us to probabilistically model each example, and thus at any point output the "current" representation. The architecture of XTRUCTURE (Fig. 5) has several layers with distributions at the foundation; tuples are fit into the model by passing them through this layered structure in a well-defined way. The layers are organized hierarchically, with each one taking care of a different aspect of the learning process. We explain each layer's role next.

The bottom layer in the hierarchy is the **symbol layer**, which holds a distribution over the ASCII characters that occur at a given position in the input tuples. This permits us to represent a position in a tuple as a character class, an or-statement, a single character, or a wildcard (".") based on the distribution. For example, if a series of mm/dd/(yy)yy dates are fed to a XTRUCTURE, the first character will hold a distribution containing only the values 1 or 0, (since $0 \leq$months≤ 12) of the year. The second will eventually converge to a uniform distribution over $[0,9]$ and thus it will be represented with the character class *digit*, *D*. We explain how we decide each representation from later in the paper.

The **token layer** represents sequences of characters from the original tuples, or *tokens*, obtained by splitting the original tuple according to a set of *delimiters*, e.g., -, /, #. The intuition is that delimiters often capture substructure of tuples. Consider the "10/1/2017" date as an example: here the three tokens are separated by */*. Each token is represented in a *token representation*, which is simply a linked list of symbols (from the symbol layer). For dates, the "months" in the date will be a token in the token layer, eventually represented as *(0—1)D*. When no delimiters are available in the data, the entire string is represented as a single token.

Tokens of different lengths cannot be represented with a single token layer. The next layer in the hierarchy, called **branch layer**, deals with variable-length data. Branch layers consist of a list of token layers, and can represent an entire tuple. In particular, a branch layer represents a list of tokens (captured by token layers) interleaved with delimiters. In our "date" example, we may find dates with two different formats for the year, i.e. 4 vs. 2 digits. These two variations will be represented with two different branches in a *branch representation*.

Each XTRUCTURE has several branch representations to represent attributes with different syntactical patterns, for example, tuples with different lengths. It is common to find dates with many different formats, due to data quality issues, as well as IDs, capitalization typos, etc.

Illustrative Example Introduction: Dates. In order to more concretely ground the methods and ideas behind XTRUCTUREs, we introduce an end-to-end running example, which we use to illustrate each component of XTRUCTURE learning. In our example, a XTRUCTURE is used to represent an date attribute that takes different formats (MM/DD/YY, M/D/YY, etc.), while permitting empty values as well as "N/A." A XTRUCTURE representing this attribute using the default set of parameters would take the form shown in 6. Later, we introduce some of the tunable hyperparameters for learning and representing the data and discuss explicitly how they affect the XTRUCTURE in our example.

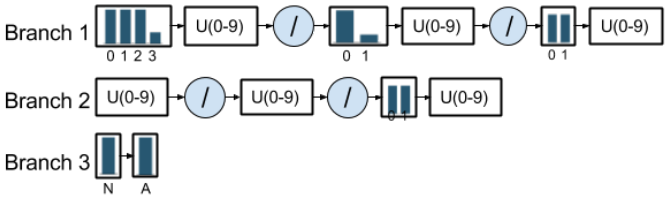

Fig. 6: Example XTRUCTURE for "date"

B. Learning a XTRUCTURE

XTRUCTUREs are adapted after each input tuple is consumed. When it gets a new tuple, XSYSTEM chooses an existing branch for the tuple, if one exists, or creates a new branch and seeds it with the input. This decision is made based on a measure of *scoring fit*. The branch representation then segments the input into tokens, K, based on a set of delimiters, and splits each token, k, into characters, updating the token and symbol layers.

The following sections describe: (1) how we compute scoring fit in Section IV-B1, (2) the branch-and-merge algorithm to support multiple branches in Section IV-B2, (3) the approach to tokenizing input tuples and feeding characters to the individual layers in Sections IV-B3 and IV-B4, and (4) an optimization to speed up learning in Section IV-C2.

1) Fitting Tuples: Scoring Fit

While learning a XTRUCTURE X, we must understand how well a tuple, t, "fits" into the structure defined by X. We introduce a scoring fit measure for this. More formally, given a tuple t and a XTRUCTURE, X, we define an operation $d(t,X) : \mathbb{R}^+$ that indicates how far t deviates from the pattern represented by X. This function is useful to fit new examples, as well as to compare XTRUCTURE, both to itself and to representations learned with other methods.

To build this function, instead of comparing each character in t to a corresponding "character" in the representation X (which is ill-defined, since our model holds a distribution over characters rather than a single character), we look at the characters S a symbol layer *represents*, and assign score $d(s_i \in S, l_i)$ for how close each s_i matches the representation in symbol layer l_i. To define d, we use GET-ASCII-CLASS(c) as the UNIX class (e.g., alphanumeric, white space, etc.) of a character c , and $l.class$ as the character class of a symbol layer l (referred to as max_class in Algorithm 2). We also define $l.is\_class$ to be a boolean indicating whether the layer's representation is its character class. This decision is based on a χ^2 test. If its p-value cannot be represented as an "OR" operation over characters, then,

$$d(s_i, l_i) = \begin{cases} 1 \text{ if GET-ASCII-CLASS}(s_i) \neq l_i.class \\ \alpha \text{ if not } l_i.\text{is\_class and } s_i \notin l_i.chars \\ 0 \text{ otherwise} \end{cases}$$

where $l_i.chars$ are the characters represented in the symbol

layer l_i, and the parameter α is used to determine how much exact character matches are prioritized compared to matches in class only (i.e. two characters). In practice, we set up $\alpha = \frac{1}{5}$ as a reasonable value for this relative weighting.

We use d to propagate the symbol layer scoring fit through a XTRUCTURE's layers, leading to a general scoring fit of a tuple with respect to the model. In particular, for token representations K, branch representations B and a modeled representation X, the distance of a tuple t to X is defined by:

$$d(t,X) = \min_{b \in B} d(t,b)$$

that is, the minimum distance of the tuple with one of the branch representations, b, of X, which is in turn defined as:

$$d(t,b) = \sum_{k_i \in b, t_i \in t} d(t_i, k_i)$$

where t_i are the tokens of the input tuple, t, that are compared with the token structures, k_i, of X as follows:

$$d(t_i,k) = \left[\sum_{l_i \in k, s_i \in t_i} d(s_i,l_i) \right] + |len(t_i) - len(k)|$$

Note the extra term in the last equation used to pad with null characters whichever is shorter between the token structure, k_i in X and the token t_i in the tuple t. This ensures XSYSTEM does not incorrectly penalize smaller valid instances of the underlying finite language, while still creating a new branch in the structure for them. For example, a column that contains several instances of "123" and one instance of "1", the latter would be padded with 2 null characters.

Illustrative Example: Scoring Tuples. Suppose we have the XTRUCTURE discussed in the introduction to our example (Figure 6). Now, we consider the scoring of three new tuples: a date "11/09/14," a misspelled date "0A/25/38," and a completely misfit value "New York." Table I illustrates how each of these three tuples are scored, and as we expect, the score is 0 for the correctly formatted string, slightly above 0 for the misspelled date, and high for the misplaced value. We also see here the effect of the α parameter, which controls how much the out-of-range year (38) impacts the overall fitting score of the string. In practice we find that XTRUCTURE is fairly robust to changes in α, but in general as is demonstrated here, increasing α penalizes out-of-distribution data more strictly.

String	Chosen Branch	Score
11/09/14	Branch 1	0
0{A}/25/[3]8	Branch 1	$1+\alpha$
N{e}{w}{ }{Y}{o}{r}{k}	Branch 3	7

TABLE I: Examples of scoring fit for the XTRUCTURE shown in Figure 6 on three example strings. {} and [] represent the errors that contribute 1.0 and α to the score respectively.

2) Representing Multiple Branches

In practice, data from the same attribute may contain values with different syntactical patterns. For example, an ID might be a 10-digit number, or simply "N/A". This phenomena inspires XTRUCTURE's multiple branch representations (that is, why we allow R to be $b-$dimensional). However, we have no way of knowing *a priori* how many different patterns are in a set of examples, a XTRUCTURE must somehow manage multiple branches, updating and representing them appropriately.

Given a new input tuple, XSYSTEM must decide whether to fit it into an existing XTRUCTURE branch, or create a new branch capture the tuple's syntactical structure. For this, we use the scoring fit. For each input t, XSYSTEM finds the "best matching" branch by doing $b_{best} = \arg\min_b d(t,b)$; if $d(t,b_{best})$ is below a *branching threshold*, the tuple is fit into that branch, otherwise a new "empty" branch is created. The existence of this branching threshold introduces the challenge of how to tune it. To avoid manually tuning such hyperparameter, we introduce an adaptive *branch-and-merge* technique.

Branch-and-Merge algorithm. The algorithm works as follows, with pseudocode shown in Algorithm 1. We hide the unintuitive data-dependent hyperparameter, and instead expose a *maximum branches* parameter (line 2), that indicates the maximum number of structures that are meant to be represented by a XTRUCTURE (b in the formal definition). This parameter can be set up based on domain knowledge, or user preference, e.g., if an analyst knows there are 3 ways of representing a business entity, he/she can choose 3 as the number of branches, as no more than those are expected to appear in the data.

Given a fixed branching threshold and the maximum number of branches desired by users, XSYSTEM proceeds as follows: for each new input, determine whether or not it "fits" within any existing branch (lines 3-4)—if so, add it to this branch, and otherwise, create a new branch (line 7). If the number of branches ever exceeds the specified maximum (line 8), we compute a pairwise distance between branches. The two closest branches b_1 and b_2 are merged by fitting generated tuples from the subsumed branch into the one subsuming (lines 9&11) – and the new "branching threshold" is set to $d(b_1,b_2)$ (line 10).

This adaptive mechanism allows XSYSTEM to correct for undershot initial thresholds, but not overshot ones, so in practice, the initial branching threshold is set to a small $\varepsilon > 0$. The entire algorithm, including both picking the best branch and branch-and-merge, is shown in further detail in Algorithm 1.

Algorithm 1: Fitting new words into XTRUCTURE

```
1  branching_threshold ← ε
2  Function learn_new_word(word: String, max_branches: Int) : void
3      best_branch ← arg min_{b∈branches} b.fit_score(word)
4      if best_branch.fit_score(word) < branching_threshold then
5          best_branch.add(word)
6      else
7          branches.add(new Branch(word))
8      if branches.length > max_branches then
           // fit(B_i, B_j) returns how well B_i fits into B_j
9          B_outer, B_inner ← arg min_{(B_i,B_j)∈branches} fit(B_i, B_j)
10         branching_threshold ← fit(B_outer, B_inner)
11         B_outer.add_word(w) ∀ w ∈ B_inner.learned_words
12         delete B_inner
```

3) Tokenization and Character Fitting

To update the token layers, the input tuple is split into tokens and then each token is fed to the layers of its corresponding token structure. The tokenizer uses special characters (delimiters) as reference for alignment. The positioning of these characters

on a string is often an indicator of data type. For example, IPv4 addresses blocks are separated by ".", while dates are usually "/" or "-" delimited.

4) Modeled Representation

During modeling, after receiving a new example and determining the token structures, each token is fed to the layers of its token structure. A symbol layer, as introduced before, holds a distribution of the characters it has seen, and represents them with their character class when is statistically significant (see Algorithm 2). Each layer is modeled as a sampling problem, under the hypothesis that every character within the majority character class is equally likely. A χ^2 test of independence is then performed, confirming or rejecting this hypothesis; if confirmed, the layer represents itself by its character class (lines 11-12 in Algorithm 2). If the null hypothesis is rejected, then there exists significant bias in the data source that should be captured in the representation, so the layer instead enumerates all fit tuples in an or-statement, in order of decreasing frequency, until a specified "capture percentage" of the distribution is captured. This corresponds to lines 14-20 of Algorithm 2. Running this process whenever a new example is encountered ensures we always model a valid XTRUCTURE.

Algorithm 2: Symbol layer representation of fitted tuples

```
1  We know all_chars_seen, and capture_threshold is a parameter
   output : A string representation of this layer
2  Function compress_layer() : String
3      class_proportions ← proportion of each character class seen
       // e.x. {"A-Z": 0.5, "a-z": 0.25, "1-9": 0.25}
4      max_class ← arg max(class_proportions)
5      max_proportion ← class_proportions [max_class ]
6      if max_proportion > 0.95 then
7          chars_to_capture ← filter(x → x ∈ max_class, all_chars_seen)
8          histogram ← histogram(chars_to_capture, bins=size(max_class))
9      else
10         chars_to_capture ← all_chars_seen histogram ←
             histogram(chars_to_capture,
             bins=sum(size(class) ∀ class ∈ class_proportions))
11     if ChiSquared(histogram) > p then
12         return max_class
13     else
14         captured ← 0
15         sort all_chars_seen by frequency
16         representation ← []
17         while captured < capture_threshold do
18             next_char ←all_chars_seen.next()
19             representation.add(next_char)
20             captured ←captured +frequency(next_char)
21         return "|".join(representation)
```

Illustrative Example: Learning Dates. Now, we return to our example of representing an attribute of "dates" with XTRUCTURE, and illustrate the algorithms outlined in this section. In order to make the example instructive, suppose that we have the partially trained XTRUCTURE given in 6; we consider now learning the same "misspelled date" as in the scoring example, and examine the effects of the tunable parameters on the final structure. In particular, we are interested in how the tunable parameters will affect the learned representation:

• **Capture Threshold:** As the capture threshold increases, we include more of the data distribution in the XTRUCTURE, at the cost of potentially including outliers; in our example, if the capture threshold is 0.9, then the data structure will likely not change after training on a single erroneous example.

• **Maximum Branches:** Our example is fairly robust to the maximum branches parameter, but it nevertheless has an effect: setting it to 1 results in a single branch recognizing dates of the form $DD/MM/YY$, which means that dates of the form $D/M/YY$ and NA are not scored, while setting the parameter extremely high results in new branches being allocated for typographical errors and outliers, which is also not desirable.

C. Optimizations

In this section, we describe several optimizations to XTRUCTUREs that help make our implementation effective in practice.

1) Parallel Learning

It is possible to learn XTRUCTURE in parallel by using the *branch-and-merge* algorithm. When fitting a model, we can use multiple workers, each one reading disjoint sets of tuples and fitting them independently. This has the benefit of exploiting the parallelism readily available in modern architectures, but leads to more than one representation per attribute. At this point, we can use the *branch-and-merge* algorithm to merge the branches of the different built models, leading to a representation equivalent to the one that a single worker would have learned.

2) Early Stopping

When learning from structured data, it is common for much of the computation time to go to fitting tuples that do not contribute to the final XTRUCTURE. Consider, for example, a long list of well-formatted dates. After a few tuples, the representation we are modeling will reflect the pattern, and will not change as additional tuples are processed.

We can *stop* learning when the model has converged and does not change after some number of new tuples are consumed. To do this, we track how much the fit of new tuples changes during fitting. Initially the scores are expected to change a lot, they will decrease and become steady over time – especially when the data is regular. The process of early stopping is inspired by an application of the Central Limit Theorem which we use also in Section V-A. The early stopping process is shown in Algorithm 3. We stop when the average fitness score tuples in the attribute drops below a threshold. To ensure confidence, our idea is to sample the distribution in groups, taking sample averages. These sample averages approximate a normal distribution around μ, the desired mean. Thus, in order to determine if the process should stop early, we generate groups of n tuples and calculate their mean fitness in the learned XTRUCTURE, as well as the standard deviation of the approximately normal distribution. We use a desired confidence of 95% to estimate the sample size (line 4).

This technique allows us to skip large amounts of data while still finding good approximate representations. The method fails when the attribute has many different branches that are seen only later. For this reason, the technique is disabled by default, and should be enabled when the user knows the data is highly regular or randomly shuffled.

Algorithm 3: Fitting tuples to branches

```
all_scores ←[]
latest_scores ←[]
Function needed_sample_size(current_std: float) : int
    return int((1.96 ∗ x/0.1)^2) // this is a normal distribution
Function new_word(word: String) : void
    if not done_adding then
        score ← ∑_{1≤i≤|layers|} layers[i].add_and_output_score(word[i])
        latest_scores.append(score)
        if len(all_scores) = 30 // Application of Central Limit Theorem
         then
            score ← avg(latest_scores)
            all_scores.append(score)
            latest_scores ←[]
        current_std ← stdev(all_scores)
        if len(all_scores)>needed_sample_size(current_std) then
            done_adding ←true
```

D. Tuple Generation and Human Readability

A XTRUCTURE needs to generate tuples that, though not necessarily part of the given examples, conform to the domain of the examples ($f(X)$, formally). This is necessary for comparison, as we see in the next section. Here, we explain how to generate tuples from a XTRUCTURE (IV-D1). Related to the generation of tuples is a string representation of XTRUCTURE which is readable by humans, a useful property to provide an overview of the data to humans which we describe in IV-D2.

1) Generating Tuples from a XTRUCTURE

To generate a tuple, XSYSTEM traverses the layers of the XTRUCTURE bottom-up. It generates *characters* through its symbol layers. These are concatenated into *tokens* by the token layer, which also takes care of interleaving the delimiters as necessary. Finally, tokens are concatenated into *branches*, and the generator selects randomly the branch that would be chosen to generate output a tuple.

To make sure each symbol layer generates characters leading to tuples that represent the structure well, instead of returning the representation of its character distribution, each symbol layer draws randomly from its corresponding character distribution, producing a string from the symbol layer. For convenience, the compress_layer function of Algorithm 2 returns such representation.

Illustrative Example: Generating Dates. Following the above algorithm, we can generate dates from our learned XTRUCTURE. Concretely, a single pass of the algorithm does the following (using Fig. 6 as example): i) choose a branch randomly (e.g., branch 1); ii) sample, for each token in the branch, the character distribution (Uniform($\{1\ldots9\}$) for Token 1 Char 1, Bernoulli(0.4)($\{0,1\}$) for Token 3 Char 1) and append the sampled characters; ii) interleave the generated tokens with the hinges, yielding the final string.

2) Making a XTRUCTURE Readable

We want to serialize a XTRUCTURE in a way that is easy to read, akin to how regexes map the underlying DFA they represent to a string. The algorithm to achieve this is similar to our tuple generation algorithm, but instead of specific tuples, we want to output the general string representation that is represented by XTRUCTURE.

When traversing a XTRUCTURE's layers bottom-up, we propagate partial representations along the way. First, the symbol layers return either an individual character, a character class (e.g. #, \w, etc.), or a group of characters depending on the result of the chi-squared test described in the previous section. Then, all the symbol layer representations of a token representation are appended, leading to a token, meaning that for a token representation k_1 with layers l_1 through l_n, where in the following $||$ represents the concatenation operator:

$$str(k_1) = \Big|\Big|_{i=1}^{n} str(l_i)$$

These token representations are then interleaved with the appropriate delimiters (kept during the learning process) to form branch representations, given that: $str(b_1) = str(k_1)||h_1||str(k_2)||h_2\ldots$. Finally, this is propagated upwards again, and the representation of an entire XTRUCTURE is simply an *OR* of all of its branches:

$$R(X) = str(b_1)||\text{“|”}||str(b_2)\ldots\text{“|”}||str(b_n)$$

Illustrative Example: Human-Readability. Following the afore-described algorithm exactly yields the following human-readable representation of our date XTRUCTURE:

$$(0|1|2|3)[0\text{-}9]/(0|1)[0\text{-}9]/(0|1)[0\text{-}9] \mid [0\text{-}9]/[0\text{-}9]/(0|1)[0\text{-}9] \mid NA$$

E. Complexity and Expressiveness Analysis

We analyze next the complexity of learning a XTRUCTURE, performing branch-and-merge, serializing the XTRUCTURE as well as matching new strings.

Complexity: Earlier, we showed that a XTRUCTURE is a DAG where each node represents a character distribution, internally implemented as a set of linked lists of character symbols. This representation supports fitness, comparison, and generation algorithms. Table III shows the time complexity of these algorithms in XSYSTEM– all algorithms in XSYSTEM are polynomial in the input size. There are three main algorithms: “Scoring”, which assigns a fit score to a candidate word as a function of how well it fits into an existing XTRUCTURE, “Branch and Merge”, which samples a data column and decides how to contract or split the XTRUCTURE when a new sample is introduced, and “Serialization”, which converts a XTRUCTURE to a human-readable and regex compatible notation. The “Scoring” algorithm is used for both building a XTRUCTURE, as well as matching a tuple against an existing XTRUCTURE, e.g., for outlier detection.

Expressiveness: The “Scoring” and “Branch and Merge” algorithms combined yield a data structure with the same expressiveness as that of DAFSA. Below we provide proofs of expressiveness XTRUCTURE w.r.t. to regular languages.

Lemma IV.1. *A* XTRUCTURE *can be converted in polynomial time to a DAFSA, and vice-versa.*

Proof: Since a XTRUCTURE is a DAG where each node represents a character distribution, a DAFSA that accepts all instances accepted by XTRUCTURE can be trivially built in

Symbol	Definition
W_d	Data column width (max tuple length).
S_d	Number of items sampled from data column.
B_x	Number of branches in the XTRUCTURE.
N_x	Number of nodes in the XTRUCTURE.

TABLE II: List of symbols used in complexity analysis.

Algorithm	Time Complexity
Scoring	$O(M_x + N_x) \equiv O(B_x * W_d)$
Branch and Merge	$S_d * (scoring + B_x * W_d)$
Serialization	$O(M_x + N_x) \equiv O(B_x * W_d)$

TABLE III: Time complexity for XSYSTEM algorithms.

polynomial time via a BFS traversal of the XTRUCTURE. Nodes either accept a single character, or any character from a "character-class". Edges in the XTRUCTURE are transitions in the resulting DAFSA. Also note that this conversion to DAFSA can be done in polynomial time because XTRUCTURE itself is deterministic, e.g. the same string never occupies more than one branch in the XTRUCTURE. ■

Theorem IV.2. XTRUCTURE *expressiveness is equivalent to the set of regular languages that can be represented by DAFSA.*

Proof: Follows from the lemma; for every XTRUCTURE there is at least one equivalent DAFSA, and vice-versa. ■

Theorem IV.3. XTRUCTURE *is equally as expressive as the finite regular languages, and is thus less expressive than DFA.*

Proof: Since XTRUCTURE is equivalent to DAFSA, and DAFSA is less expressive than DFA, it follows that XTRUCTURE is necessarily less expressive than DFA. Specifically, XTRUCTURE cannot *minimally* represent regular languages that contains cycles. ■

Note that we are not interested in learning minimal DFA in XSYSTEM. Indeed, even if we had chosen a data structure that has the same expressiveness as that of DFA (e.g., it allows cycles), there is no polynomial time algorithm guaranteed to produce a DFA of size at most polynomially larger than the smallest consistent DFA using only positive samples [16].

In practice, we also do not need to learn minimal DFAs here because our positive samples are drawn from highly structured data, and instances of each language are finite, e.g., emails, telephone numbers, and chemical identifiers. The expressiveness of DAFSA alone is quite powerful and covers all of our finite languages use cases, while also doing a good job at situations where a dataset attribute is not finite and the user only cares about tuples up to a certain size. For example, assuming a dataset attribute is captured by a small cyclic DFA, but we are only interested in instances of length at most k, a DAFSA that represents this finite subset of the original language, and that is at most $k+1$ states larger than the DFA, can be obtained in polynomial time.

V. COMPARING XTRUCTURES

In this section, we explain how to measure similarity between XTRUCTUREs (V-A1) as well as how to it efficiently in V-B.

A. Measuring Similarity for Comparison

The comparison operation of XSYSTEM relies primarily on the scoring fit defined in the previous section and the Central Limit Theorem, as we explain below.

1) Comparing with other XTRUCTUREs

We want a *syntactic distance* function between the structure represented by different XTRUCTURE, such as $D(X_1, X_2) : \mathbb{R}+ \times \mathbb{R}+$, that returns a pair of scores between 0 and 1 representing how well the structure of each XTRUCTURE "fits" into the other. Previously, in Section IV-B1, we discussed a *scoring fit* obtained when fitting a tuple to a XTRUCTURE. We define now the fit of a XTRUCTURE, X_1 *into* X_2 as the average scoring fit of the set of tuples represented by X_1 that fit X_2.

In general, it is infeasible to generate *all* possible tuples represented by a XTRUCTURE. Instead, we model $D(S, X_2)$ as a distribution for which we want to estimate the mean fit with a certain degree of confidence. This reduces the problem from one of generating all tuples, to one of generating a subset of tuples that will allow us to estimate the mean fit in a statistically significant manner. However, to reliably estimate the mean fit we would need the underlying distribution of the data, which we do not know. We also do not want to make assumptions about this distribution: it will be multi-modal at best, and completely irregular at worst.

To address this, we use the central limit theorem as in Section IV-C2; sampling the distribution in groups approximates a normal distribution around μ, the desired mean. Thus, in order to compute the fit of X_1 in X_2, we generate groups of n tuples from X_1 and calculate their mean and standard deviation of fit into X_2; we use a 95% confidence interval to estimate the sample size.

XTRUCTUREs as Proxy for Column Comparison. We show experimentally in Section VI-B that for a reasonable distribution of data and selection of parameters, XTRUCTUREs can act as a proxy for attributes in terms of comparison. In particular, XSYSTEM uses XTRUCTUREs to effectively compare attributes and identify those that come from the same underlying distribution. More formally, the XTRUCTURE comparison score can be viewed as modified expected Hamming distance aligned at hinges. In this distance function, index characters are sampled independently at each branch, with probability $P(\text{char}|\text{branch} = \{1, \ldots, \text{max-branches}\})$.

2) Comparing with regexes

When comparing the structure represented by a XTRUCTURE, X with one represented by a regex, R, we also want to obtain a tuple of scoring fits: how well X fits R and the other way around. As with the comparison process between XTRUCTUREs, our approach involves generating tuples from the XTRUCTURE (or regex), and then measuring how well the generated tuples fit the regex (or XTRUCTURE). The difference from the approach in the previous section is that tuple values are binary, i.e., either X fits R or it does not (and vice versa).

To compute the similarity between a XTRUCTURE a regex, we can calculate the probability of the structure held in a

XTRUCTURE, X, fitting a regex, R, as follows:

$$P(fit(X,R)) = \sum_{n=1}^{N} match(g(X,n),R)/N$$

where the function $g(X,n)$ generates a tuple from X and the function *match* returns 1 if a tuple fits R and 0 if it does not. The total number of draws, N is chosen through standard application of the CLT, which allows us to treat this as estimation of $\hat{P_{fit}}$, a Bernoulli random variable, and therefore get an approximation within a certain range and confidence interval.

To compute the fitness of R with respect to X, we use existing libraries that produce strings from existing regular expressions, commonly known as *xeger*. Using one of these *xeger*-like tools, we generate tuples from the regular expression and then we apply the same technique in the opposite direction.

We use this approach to compare XTRUCTURE with already existing regular expressions for our automatic label assignment application. We obtain good results that we present in the evaluation section. However, it is worth noting a few limitations of the approach with respect to the XTRUCTURE-XTRUCTURE comparison method.

First, if a regex is too specific the similarity with a nearby structure may be counter-intuitively low. For example, the structure of a regex that represents exactly *"ABCD"* will have a low similarity to a XTRUCTURE's structure that represents *"ABCE"*, while this would not be the case if the two structures to be compared would be represented by XTRUCTUREs.

Second, due to our need to generate tuples from the regular expression, the regex must be finite, so wildcard characters are not allowed. Although seemingly limiting, this is not a great disadvantage, as *highly structured* tuples will tend to lack wildcard characters – which indicates a lack of structure.

Why not compare regexes with the original data directly? A natural question is why do we compare a XTRUCTURE with a regex instead of comparing the original data directly to the regex. There are three key advantages to our approach. First, it is easy to sample from a XTRUCTURE, as it already represents the branches in the underlying data. The alternative would be to perform expensive random sampling on the data directly, which is difficult if we want to sample from all the possible syntactical variations. Second, sampling from XTRUCTURE involves generating tuples in-memory and feeding them in streaming to the XTRUCTURE, as opposed to accessing and reading data from a data source. This is especially beneficial because we need to repeat this operation every time a new regex is added to the library, which happens often when multiple analysts participate in the process. Last, it is more convenient to compare the XTRUCTURE learned by XSYSTEM to the regexes as comparisons can naturally be parallelized, and once the XTRUCTURE is learned, it is readily available to be used with other applications.

B. Efficient Large Scale Comparison

Recall that one of our applications is to find which attributes are syntactically similar. Naively, this entails performing an all-pairs comparison of XTRUCTURE, an $O(n^2)$ operation that becomes prohibitively expensive in settings with large numbers of attributes. We rely on an approximate technique based on locality-sensitive hashing (LSH) [17] with minhash signatures.

Adapting LSH with minhash to XTRUCTURE is challenging because we do not have sets of elements, but XTRUCTUREs that can generate them. The XTRUCTURE, however, does not generate sets of tuples deterministically, and the space of tuples it represents can be very large, making it difficult to generate good minhash signatures. In addition, instead of estimating the syntactic similarity of XTRUCTURE we would be just estimating the similarity of the sets of tuples they generate, which is not what we want. For this method to work, we need a way of generating minhash signatures from XTRUCTUREs *deterministically* and in a way that captures the syntactic features learned during the building process.

We solve this by generating triples of the form (character, last_hinge, index). The first element represents the character or character class, the second one is used to determine the token of which the character is part, and the last one is the position of the character within the token. Codifying all this information in triples preserves the structural information in a way that allows us to still employ minhash. For example, for the string *AB;CD*, we would generate the set (A,0,0),(B,0,1),(C,1,0),(D,1,1)). With the set available, we then use minhash to obtain a signature.

In our evaluation we show that this method greatly reduces the comparison runtime, with a minor reduction in accuracy.

VI. EVALUATION

In this section, we look at the performance of XSYSTEM and study how it helps address our motivating applications. Using a range of real datasets and workloads we (1) study how XSYSTEM can propagate labels from annotated regexes to columns in the datasets (VI-A); (2) use XSYSTEM to learn XTRUCTUREs on columns of a dataset, and use these XTRUCTUREs to identify syntactically similar content (VI-B); and (3) use XSYSTEM to detect syntactical outliers from real data (VI-C). We also conduct a series of microbenchmarks to understand the performance of XTRUCTURE (VI-D).

Datasets and setup. We use the following datasets: i) **university data warehouse (DHW)** which consists of 161 tables and 1690 attributes with information about departments within the university; ii) **ChEMBL (CHE)**, a public chemical database [18] with 70 tables and 461 attributes; iii) **data.gov (GOV)**, US open government data, consisting of 2250 CSV files; and iv) **MassData (MAS)**, the open government data from Massachusetts, from which we use 10 attributes for our outlier-detection experiment. For the outlier-detection experiments we also use the KDDCUP99 and Forest Cover datasets [19], which are standard datasets used in outlier detection. Unless otherwise specified we set maximum branches to 3, the branching threshold to 0.1, and the capture threshold to 85%. For all the single-threaded experiments (all except as indicated), we use a computer with a 1.7GHz Intel Core i7 and 8GB RAM.

A. Automatic Label Assignment

To automatically label columns, we need a pre-built library of (regex, label) pairs that associate meaningful labels to the

syntactic patterns described by the regexes. Given such a library (which works as well as ground truth), we can use XSYSTEM to learn syntactic patterns for each column in the database and compare these patterns with the regexes in the library. Then, when we find a syntactic match between a XTRUCTURE and regex, we assign the label to the column represented by that XTRUCTURE. The quality of this application depends on the quality of our comparison technique, which we evaluate here.

To obtain the library of (regex, labels), we manually assigned (regex, label) pairs to more than 4,000 attributes from DWH, CHEM and GOV. The specific number of attributes with assigned labels is shown in the "# total attrs." column of table IV. The regexes are drawn from regexlib.com [11], which has a collection of generic patterns. We also added domain-specific regexes for chemical datasets. In both cases we choose the most specific regex possible. For example, for an attribute containing even numbers, we would use "\d\d(0|2|4|6|8)" rather than "\d\d\d".

We used XSYSTEM to learn the XTRUCTUREs for the 4000+ attributes and searched for the nearest regex in the regex library, using the algorithm of section VI-B. We compared this nearest regex to the ground truth regex we manually associated with each attribute. Table IV shows that we find over 94% of correct matches for the three datasets we use. This means that we can automatically assign labels to 94% of the data, which vastly reduces the human effort that would otherwise be necessary.

Dataset	total attrs.	correct matches	% matches
DWH	1504	1417	94%
CHE	307	294	95.5%
GOV	2476	2355	94.9%

TABLE IV: XTRUCTURE-regex correct matches vs. dataset.

Not all matches are equally useful. For example, we find matches of columns to both InchI numbers and keys as well as to SMILES, and both of are annotated with *chemical id*. This vastly improves the discoverability of these attributes, helping analysts with their tasks. In other cases, the match is with a low specificity regex such as "strings" or "numbers", which although correct is not insightful. This is an artifact of the quality of the (regex, label) pairs we had available. In the enterprise scenario, we expect registries of regexes built by domain experts to be of high quality, therefore leading to good quality label annotation of the data.

In summary, this experiment shows that **XSYSTEM is able to propagate labels to attributes for a wide range of attribute formats when a library of (regex, label) is available.** This is provided that the regex in the library are of good quality and that XTRUCTURE are specific enough, an aspect we evaluate in the microbenchmarks section.

B. Summarization and Comparison

In this experiment, our goal is to use XSYSTEM to find pairs of syntactically similar attributes in a large dataset; such columns often represent duplicates, or possible identifiers that can be used in joins. For this application, we obtained ground-truth data consisting of pairs of syntactically similar attributes from CHE. We collect all attributes whose name contains "id" (e.g. "tid," "cell_id," "tax_id"); attributes were removed and tuples shuffled in random order; a volunteer labeled pairs of these shuffled nameless attributes as syntactically similar/different. We obtained labels for about 1000 pairs of attributes. We learn XTRUCTUREs for each attribute.

We evaluate the effectiveness of XSYSTEM at finding syntactically similar pairs. First we perform an all-pairs comparison between the learned XTRUCTUREs using the method described in V-A1. This is $O(n^2)$ but is an intuitive method, useful when the number of attributes is small, or when we want to quickly find all IDs in a database that are syntactically similar to one pre-selected column ID. In this experiment, the method labels a pair of columns as syntactically similar when their similarity is above a given threshold, and then we measure precision and recall of the results, which we show as the "All Pairs" line in Fig. 9 (left). The figure shows a good accuracy, with the method reaching an F1 score of around 0.82, and maintaining constant high precision until a recall of about 0.8.

Fast Comparison. Because all-pairs comparison becomes expensive with thousands of attributes, we implemented the approach of section V-B. When using this approach, XTRUCTUREs are clustered based on the approximate Jaccard distance between their signature sets. These clusters were then translated into pair labellings, giving an $O(n)$ time algorithm. The precision and recall results for the same experiment using this method is shown on the "MinHash LSH" line of Fig. 9 (left). The figure shows that the quality of this alternative method is in fact similar to the all-pairs one, with the curve shapes looking similar. The slight irregularities in the curve (lack of smoothness) at high recalls are due to the cluster-based nature of the LSH method, rather than direct comparison of each pair of attributes. This makes sense because since we must pre-generate strings to generate the MinHash signature, we make sure the strings uniformly represent the underlying data, therefore increasing the signature quality. We further explore the details of the performance tradeoff of these two methods in the microbenchmark in section VI-D.

Qualitative Analysis. When the techniques yield errors, we found them to be quite intuitive. For example, one common error we found was due to irregularities in the data, such as two similar attributes not being detected because one used "nan" to denote missing data, while the other used "-1". Another common kind of error came from attributes with diverse representations and implicit semantic meaning. For example, a human may label two attributes containing variable length decimal numbers as different if the mean or standard deviation of the numbers is different, which, in some cases XSYSTEM is not able to detect, yielding false positives.

C. Syntax-Based Outlier Detection

Next, we evaluate XSYSTEM's ability to detect outliers within single attributes in a dataset. We use both the MAS dataset (for which ground truth was manually collected through volunteers), as well as the KDDCUP 1999 intrusion detection dataset and the Forest Cover dataset (obtained from [19]). For quantitative analysis, we use three of the outlier types from KDDCUP, as well as the Forest Cover dataset; we then utilize the manually

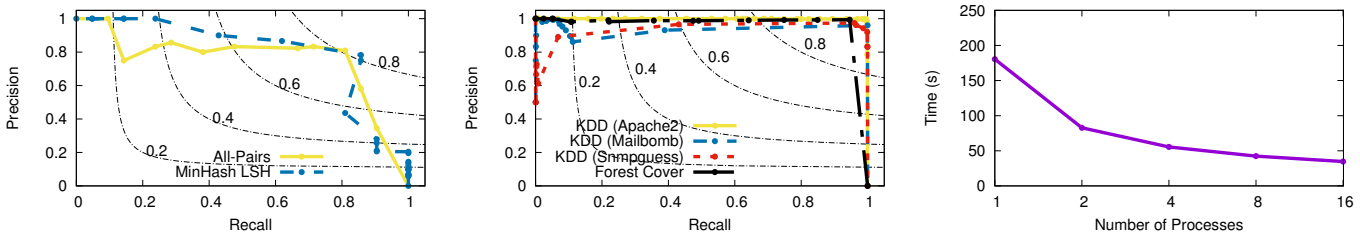

Fig. 7: Left: PR curve for All-Pairs and LSH; Center: PR curve for outlier experiment; Right: Scalability microbenchmark

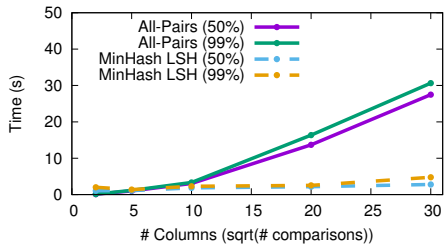

Fig. 8: All-Pairs vs MinHash LSH methods

labeled MAS for qualitative discussion.

We learn a XTRUCTURE per attribute from a subset of the tuples (with outliers present). We then freeze the XTRUCTURE and feed it new tuples, obtaining a fitness score which we use to find outliers. The score is transformed into an *outlier score* using a weighted combination of the scoring fits (section IV-B1) of each branch. We mark outlier scores that exceed an *outlier threshold*. We present the resulting PR curves in Fig. 9 (center).

The figure shows the precision and recall for different values of the *outlier threshold*. The results show near-perfect performance on all four of the large datasets. Since XSYSTEM excels with large quantities of structured data, we next perform a qualitative analysis of outlier detection using MAS, a smaller but more complex real dataset where outlier marking can actually be quite subjective; this allows us to identify the areas where XSYSTEM has the most difficulty.

Qualitative Analysis on MAS dataset. Many of the errors made by XSYSTEM are ambiguous to humans; in particular, the majority of the mistakes made were in an attribute representing street address suffixes (ST, BL, AV, etc.). The source of ambiguity is lower-frequency, but still valid street suffixes, such as "BL" for boulevard, or "PL" for place, and whether or not XSYSTEM marked these as outliers is simply a function of the aforementioned outlier threshold. On the positive side, the system found outliers that were indeed errors in the data, such as a ZIP code in Boston with more than 8 digits (the standard is 5 digits), or the tuple *MIDNIGHT* among tuples representing hours as digits.

D. Microbenchmarks

Learning Speed. Properties such as number of tuples, number of delimiters per tuple and pattern heterogeneity affect the performance of XSYSTEM. To measure these effects, we generated synthetic data with varying properties and then we ran XSYSTEM on the data. Fig. 9(a, b and c) shows the running times averaged 10 times, with median, 95th and 99th percentile.

In the first experiment (a), we use a fixed length data types (country currency codes) and vary the number of tuples in the input dataset. In the second experiment (b), we measure the impact of the number of hinges, which has an effect on the number of tokens that XSYSTEM must maintain. Here, we create input datasets with variable number of hinges by concatenating YYYY-MM-DD formatted dates, and fixed the number of tuples to 1000. Finally, in the third experiment (c), we vary attribute value length using datasets with 1000 tuples and mixed data types, which has an effect on the total number of branches that per XTRUCTURE that XSYSTEM maintains.

XSYSTEM's performance in all 3 experiments grows linearly with the variable of interest. Although absolute numbers are higher in the third experiment, 99th percentile is below 10 seconds and median below 1 second.

Studying XTRUCTURE Specificity. The specificity of XTRUCTURE is determined by how aggressively it represents the fed samples. For example, a XTRUCTURE with `"(1|2|3|4|5)"` is more specific than `"(D)"`. Layer compression is in turn controlled by the *p-value* used during the Chi control performed in line 11 of Algorithm 2.

To understand the impact of p-value on similarity to a target regex, we vary its value and compare the fitness score of each resulting XTRUCTURE against the regex. We run this experiment over 1000 tuples of a "date" attribute. The regex captures only valid instances of dates. Fig. 10 shows that the higher the p-value, the longer the final XTRUCTURE is, indicating a more specific fit. This makes sense: the higher the p-value, the smaller the confidence interval and the more likely the system is to treat sampled tuples as representative of the underlying distribution. Hence, XSYSTEM is more likely to include these samples in the learned XTRUCTURE as additional branches.

In general, the fitness score is higher for larger (more specific) p-values, and lower for smaller values. As an example, when p-value is the lowest (least specific), the learned XTRUCTURE is `DDDD-DD-DD`. The fitness score of a XTRUCTURE against a regular expression is calculated by drawing samples from it, and feeding them into the regular expression. In the case of `DDDD-DD-DD`, even though the data only contained valid dates, samples drawn from this XTRUCTURE may not necessarily be a valid date e.g., `1234-56-78`. So the less overfit the XTRUCTURE, the higher its recall and the lower its precision.

Parallel Scalability. To understand the parallel scaling of XSYSTEM, we generated data (about 20000 alphanumeric identifiers) and learned a XTRUCTURE using a different number of cores. We use XSYSTEM with *max_branches* set to 7 and measure the time the learning process takes. Since the

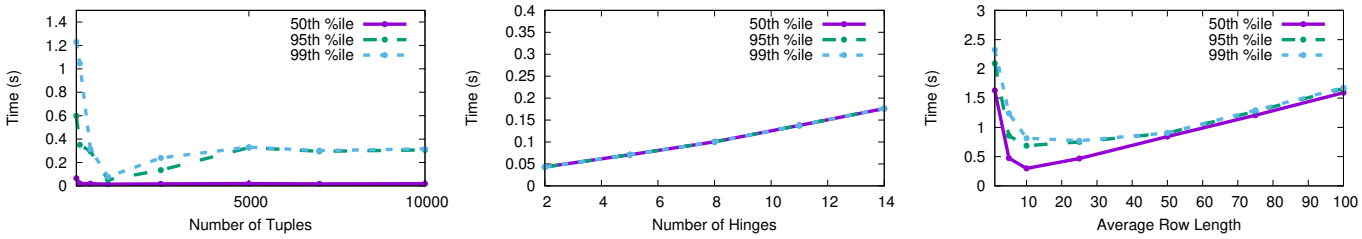

Fig. 9: Performance microbenchmarks using synthetic datasets

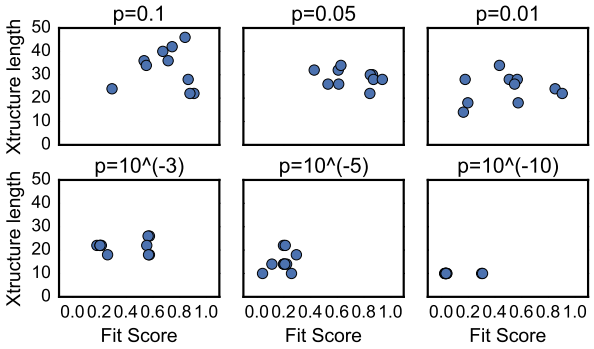

Fig. 10: P-value and fit-score effect on XTRUCTURE specificity

data is highly regular and XSYSTEM only consumes a few samples before achieving 95% confidence and stopping early (as described in section in IV-C2), we disable early stopping in order to accurately demonstrate the effects of parallelization.

Fig. 9 (right) shows our results. As expected, adding parallelism reduces the total runtime up to the maximum number of hardware cores available in the experimental machine. The system does not scale perfectly linearly after 8 cores due to overheads during the merging stage of our algorithm, which could further be reduced through optimizations, including hierarchical parallelization of the merging operation itself.

All-Pairs vs. LSH Comparison. We want to understand the runtime difference between the All-Pairs method and the LSH one. We generate datasets of uniform column length, but with varying numbers of columns. XTRUCTUREs are learned for each column (untimed), and then every pair of columns is compared using both the all-pairs method and the MinHash LSH method. The experiment of Fig. 8 shows that for up to about 10 columns (which corresponds to on the order of 100 comparisons), the all-pairs method outperforms LSH due to its low overhead. However, for larger numbers of attributes, the LSH method is superior in that it scales linearly.

Invariance to Tuple Ordering. Finally, we wish to measure the invariance of XTRUCTURE to the random shuffling of tuples. To do this, we take three different synthetic attributes of various complexities, representing IP Address, Title, and Latin Word. Each attribute contains 1000 tuples, and these are shuffled in 20 distinct ways. Table V indicates the variation across the unique shufflings, both in fitting time, and in the "fitness score" against the source column. The results show robustness against bad orderings of tuples within an attribute.

VII. CONCLUSIONS

XSYSTEM learns XTRUCTURE, which represent the syntactic structure of data. XTRUCTURE are less expressive than regexes,

	IP Addresses	Titles	Latin Words
Example	159.112.55.237	Dr.	cupiditate
Mean (ms/line)	0.27	0.23	1.0
Stdev (ms/line)	0.07	0.05	0.3
Mean score	0.18	0.25	0.41
Stdev score	0.02	0.06	0.12

TABLE V: Avg & std.dev learning time, fit with random shuffling

but orders of magnitude faster to learn, and expressive enough to represent the highly structured data often found in databases. We demonstrated XSYSTEM with 3 applications of interest for data discovery.

VIII. ACKNOWLEDGEMENTS

We thank MIT IS&T for providing access to the DWH dataset.

REFERENCES

[1] S. Heller, A. McNaught *et al.*, "InChI - the worldwide chemical structure identifier standard," *Journal of Cheminformatics*, 2013.
[2] A. Bartoli, A. De Lorenzo *et al.*, "Inference of Regular Expressions for Text Extraction from Examples," *IEEE TKDE*, 2016.
[3] F. Brauer, R. Rieger *et al.*, "Enabling Information Extraction by Inference of Regular Expressions from Sample Entities," in *CIKM*, 2011.
[4] Y. Li, R. Krishnamurthy *et al.*, "Regular Expression Learning for Information Extraction," in *EMNLP*, 2008.
[5] H. Fernau, "Algorithms for Learning Regular Expressions from Positive Data," *Information and Computation*, 2009.
[6] G. J. Bex, F. Neven *et al.*, "Inference of Concise Regular Expressions and DTDs," *TODS*, 2010.
[7] M. Lee, S. So *et al.*, "Synthesizing Regular Expressions from Examples for Introductory Automata Assignments," in *GPCE*, 2016.
[8] J. K. Feser, S. Chaudhuri *et al.*, "Synthesizing Data Structure Transformations from Input-output Examples," *PLDI*, 2015.
[9] V. Le and S. Gulwani, "FlashExtract: A Framework for Data Extraction by Examples," in *PLDI*, 2014.
[10] R. Singh, "Blinkfill: Semi-supervised programming by example for syntactic string transformations," *PVLDB*, 2016.
[11] regexlib, "Regular expression library," http://www.regexlib.com, 2016, [accessed 15 Jan 2018].
[12] A. Arasu *et al.*, "Efficient Exact Set-similarity Joins," in *VLDB*, 2006.
[13] V. Satuluri and S. Parthasarathy, "Bayesian Locality Sensitive Hashing for Fast Similarity Search," *VLDB*, 2012.
[14] Z. Abedjan, X. Chu *et al.*, "Detecting Data Errors: Where Are We and What Needs to Be Done?" *VLDB*, 2016.
[15] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, 1987.
[16] L. Pitt *et al.*, "The Minimum Consistent DFA Problem Cannot Be Approximated Within Any Polynomial," *J. ACM*, 1993.
[17] A. Gionis, P. Indyk *et al.*, "Similarity Search in High Dimensions via Hashing," in *VLDB*, 1999.
[18] A. Gaulton, L. J. Bellis *et al.*, "ChEMBL: a large-scale bioactivity database for drug discovery," *Nucleic Acids Research*, 2012.
[19] M. Lichman, "UCI machine learning repository," 2013.