Title
FlashProfile: Interactive Synthesis of Syntactic Profiles.

Journal
CoRR, abs/1709.05725

Authors
Padhi, Saswat
Jain, Prateek
Perelman, Daniel
et al.

Publication Date
2017

Peer reviewed

FlashProfile: Interactive Synthesis of Syntactic Profiles

Saswat Padhi*
UCLA
padhi@cs.ucla.edu

Prateek Jain
Microsoft Research
prajain@microsoft.com

Daniel Perelman
Microsoft
danpere@microsoft.com

Oleksandr Polozov
University of Washington
polozov@cs.washington.edu

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Todd Millstein
UCLA
todd@cs.ucla.edu

ABSTRACT

We address the problem of learning comprehensive syntactic profiles for a set of strings. Real-world datasets, typically curated from multiple sources, often contain data in various formats. Thus any data processing task is preceded by the critical step of data format identification. However, manual inspection of data to identify various formats is infeasible in standard big-data scenarios.

We present a technique for generating comprehensive syntactic profiles in terms of user-defined patterns that also allows for interactive refinement. We define a syntactic profile as a set of succinct patterns that describe the entire dataset. Our approach efficiently learns such profiles, and allows refinement by exposing a desired number of patterns.

Our implementation, FlashProfile, shows a median profiling time of 0.7 s over 142 tasks on 74 real datasets. We also show that access to the generated data profiles allow for more accurate synthesis of programs, using fewer examples in programming-by-example workflows.

1. INTRODUCTION

In modern data science, most real-life datasets lack high-quality metadata — they are often incomplete, erroneous, and unstructured [15]. This severely impedes data analysis, even for domain experts. For instance, a merely preliminary task of *data wrangling* (importing, cleaning, and reshaping data) consumes 50 – 80% of the total analysis time [33]. Prior studies show that high-quality metadata not only help users clean, understand, transform, and reason over data, but also enable advanced applications, such as query optimization, schema matching, and reverse engineering [47, 38]. Traditionally, data scientists manually inspect samples of data, or create aggregation queries. Naturally, this approach does not scale to modern large-scale datasets [38].

Data profiling is the process of automatically discovering useful metadata (typically as a succinct summary) for the data [6]. In this work, we focus on *syntactic* profiling, i.e. learning a succinct structural description of the data. We present FlashProfile, a novel technique for learning syntactic profiles that satisfy the following three desirable properties:

Comprehensive: We expose the syntactic profile as a set of patterns, which cover 100% of the data.

Refinable: Users can interactively refine the granularity of profiles by requesting the desired number of patterns.

Extensible: Each pattern is a sequence of atomic patterns, or *atoms*. Our pattern learner $\mathscr{L}_P$ includes a default set of atoms (e.g., digits and identifiers), and users can extend it with appropriate domain-specific atoms for their datasets.

Birth Year
1900
1877
1860
?
1866
1893
⋮
1888 ?
1872

(a) Dataset

Automatically suggested profile:
- “1” ◇ Digit$^{\times 3}$ (2 574)
- “?” (653)
- Empty (80)
- “18” ◇ Digit$^{\times 2}$ ◇ “?” (16)

Profile learned on requesting 5 patterns:
- “18” ◇ Digit$^{\times 2}$ (2 491)
- “?” (653)
- Empty (90)
- “190” ◇ Digit (83)
- “18” ◇ Digit$^{\times 2}$ ◇ “?” (16)

(b) Suggested and refined profiles

Figure 1: Profiles generated by FlashProfile for a real dataset[1]. Atoms are concatenated with ◇, and superscripts indicate their repetitions. The number of matches for each pattern is shown on the right.

Example: Figure 1 shows the syntactic profiles generated by FlashProfile for an incomplete and inconsistent dataset containing birth years. The profiles expose rare patterns in the data that are otherwise hard to notice. For example, the automatically suggested profile in Figure 1(b) reveals dirty years of the form “18” ◇ Digit$^{\times 2}$ ◇ “?” which constitute less than 0.5% of the dataset. As shown at the bottom of the figure, if a user requests one more pattern to refine the profile, FlashProfile separates the “18” ◇ Digit$^{\times 2}$ years from the much sparser “190” ◇ Digit years.

The closest related tools that learn rich syntactic patterns are Microsoft SQL Server Data Tools [2] (SSDT), and Potter's Wheel [43]. Beside the lack of support for refinement, neither tool generates comprehensive profiles. For the data in Figure 1(a), SSDT generates[2] the following profile, which omits years matching “18” ◇ Digit$^{\times 2}$ ◇ “?”:

- \d\d\d\d (79%)
- ? (20%)

*Work done during an internship at Microsoft.

[1] Linda K. Jacobs, The Syrian Colony in New York City 1880-1900. Accessed at http://bit.ly/LJacobs

[2] Using “Column Pattern Profiling Task” in SSDT 14.0.61021.0 in Visual Studio 2015 with PercentageDataCoverageDesired = 100.

Similarly, Potter's Wheel only detects the most frequent pattern in the dataset. Furthermore, for our example from Figure 1(a), it detects the most frequent pattern as Int but not its fixed length 4. We give a detailed comparison in § 8.

Our Technique: We profile a given dataset by first partitioning it into syntactically similar clusters of strings, and subsequently learning a succinct pattern describing each cluster. To facilitate user-driven refinement of the results into more clusters, we construct a hierarchical clustering over the dataset. This enables efficient extraction of clusters with desired granularity by splitting the hierarchy at an appropriate height. Two major challenges to constructing the hierarchy are — **(1)** defining an appropriate dissimilarity measure that that allows domain-specific profiling, and **(2)** computing all pairwise dissimilarities, which is typical for hierarchical clustering, is expensive for large datasets.

Our key insight towards addressing the first challenge is that, the desired measure of dissimilarity is not a property of the strings per se, but of the patterns describing them over the user-defined language. We define syntactic dissimilarity based on costs of patterns – a low cost pattern describing two strings indicates a high degree of syntactic similarity. To address the second challenge, we show sampling and approximation techniques which reuse previously learned patterns to approximate unknown pairwise dissimilarities. This enables hierarchical clustering using very few pairwise dissimilarity computations. In essence, we present a general framework for profiling, based on an efficient hierarchical clustering technique which is parameterized by a pattern learner $\mathscr{L}$, and a cost function $\mathcal{C}$ over patterns.

Implementation and Evaluation: Our implementation, FlashProfile, uses a pattern learner based on *inductive program synthesis* [32] – an approach for learning programs over an underlying domain-specific language from an incomplete specification (such as input/output examples). We formally define the synthesis problem for our pattern language $\mathcal{L}_P$, and present **(1)** a *sound and complete* pattern learner $\mathscr{L}_P$ over a user-specified set of atoms, and **(2)** a cost function $\mathcal{C}_P$ over $\mathcal{L}_P$ patterns. We have implemented $\mathscr{L}_P$ using PROSE [4] (also called FlashMeta [40]), a state-of-the-art inductive synthesis framework.

We evaluate our technique on 74 publicly-available real datasets[3] collected from online sources. Over 142 tasks, FlashProfile achieves a median profiling time of 0.7s, 77% of which complete in under 2s. Apart from being refinable interactively, we show that profiles generated by FlashProfile are more expressive compared to three state-of-the-art existing tools, owing to its extensible language.

Applications: The benefits of comprehensive profiles extend beyond data understanding. An emerging technology, programming by examples [32, 19, 21] (PBE), provides end users with powerful semi-automated alternatives to manual data wrangling. A key challenge to its success is finding a representative set of examples which best discriminate the desired program from a large space of possible programs [37]. We show that FlashProfile helps existing PBE systems by identifying syntactically diverse inputs.

We have investigated 163 scenarios where Flash Fill [18], a popular PBE system for string transformations, requires > 1 example to learn the desired transformation. In 84% of them, the representative examples belong to *different syntactic clusters* identified by FlashProfile. Moreover, for 86% of them, an interaction guided by the profile completes the task in a *minimum possible* number of examples.

[3] Available at: https://github.com/SaswatPadhi/ProfilingExperiments

In summary, we present the following major contributions:

- (§ 3) We define interactive profiling as a problem of hierarchical clustering based on syntactical similarity, followed by qualifying each cluster with a *pattern*.
- (§ 4) We propose a novel dissimilarity measure which is superior to traditional string-similarity measures for estimating syntactic similarity. We also present sampling and approximation strategies for efficiently constructing hierarchies using the proposed measure.
- (§ 5) We instantiate our technique as FlashProfile, using program synthesis for learning patterns over the language $\mathcal{L}_P$. Our learner $\mathscr{L}_P$ also supports user-defined patterns.
- (§6) We evaluate FlashProfile's performance and accuracy across 142 tasks on real-life datasets, and compare profiles generated by FlashProfile to state-of-the-art tools.
- (§ 7) We show how FlashProfile helps PBE systems by identifying a representative set of examples for the data.

2. MOTIVATING SCENARIO

In this section, we discuss a practical data analysis task and show the benefit of comprehensive data profiles. Consider the task of gathering descriptive statistics (e.g. range and mode) of the data in Figure 1(a). The following Python script is a reasonable first attempt after a quick glance on the data:

```
clean = [int(e) for e in years if e != '?']
min_year = min(clean)
max_year = max(clean)
mode_year, mode_count = Counter(clean).most_common(1)[0]
```

But it fails when `int(e)` raises an exception on `e == ''`. Thus, the user updates the script to clean the data further:

```
clean = [int(e) for e in years if len(e) == 4]
```

Having encountered only 4-digit years and `?` in the first sample, she simplifies the script to consider only clean 4-digit years. Now the script runs to its completion and returns:

```
min_year = 1813        mode_year = 1875
max_year = 1903        mode_count = 118
```

However, the `mode_count` value is incorrect because the analysis script ignores any approximate entries with a trailing `?` (e.g., `1875?`). Since the user is unaware of such entries after a quick glance on the data, she does not realize that the computed value is incorrect.

A more cautious analyst might have discovered entries like `1875?` after adding a second check to filter out the missing entries. However, such trial-and-error pattern discovery requires several time-consuming iterations to cover the entire dataset. The analyst also needs to manually inspect the failing cases and find a general pattern to handle them properly. Moreover, this approach works only if she knows how to validate the data type (note the `int(e)` in `clean`), which might be difficult for non-standard types.

Wrangling by Examples: Instead of manually analyzing the data, the user may choose a semi-automated technique like Flash Fill. She loads her data in Excel and provides a few examples, expecting Flash Fill to learn the transformation for cleaning any dirty years by extracting their integer parts. Typically users provide examples for the first few rows. Flash Fill then synthesizes the *simplest generalization* [18] over the provided examples.

func PROFILE$_{\langle\mathscr{L},\mathcal{C}\rangle}$($\mathcal{S}$: String[], m: Int, M: Int, θ: Double)
Result: $\widetilde{P}$, a set of patterns with partitions, such that $m \leqslant |\,\widetilde{P}\,| \leqslant M$.
1 · $H \leftarrow$ BUILDHIERARCHY$_{\langle\mathscr{L},\mathcal{C}\rangle}(\mathcal{S}, M, \theta)$
2 · $\widetilde{P} \leftarrow \emptyset$
3 · **for all** $X \in$ SPLIT(H, m, M) **do**
4 · $\quad P \leftarrow$ LEARNBESTPATTERN$_{\langle\mathscr{L},\mathcal{C}\rangle}(X)$.Pattern
5 · $\quad \widetilde{P} \leftarrow \widetilde{P} \cup \{\langle$Data: X, Pattern: $P\rangle\}$
6 · **return** $\widetilde{P}$

Figure 2: Our profiling algorithm, using a pattern learner $\mathscr{L}$ and cost function $\mathcal{C}$ over patterns.

In this case, Flash Fill produces *identity transform* since the first several inputs are all clean 4-digit years. Fixing this mistake requires manual inspection, which reveals a discrepancy only at the 84$^{\text{th}}$ row. Furthermore, if the user invokes the `MIN`, `MAX`, or `MODE` functions on the output column, Excel silently ignores all non-numeric entries, thus producing incorrect results without any warnings.

As we show in § 7, a profile of the input column allows Flash Fill to learn the desired transformation with fewer examples. Instead of simply taking examples in the order they appear in the dataset, Flash Fill proactively requests it on the *most discrepant* row w.r.t. the previous examples.

3. OVERVIEW

In this section, we informally explain our technique, and its instantiation as FlashProfile. The goal of syntactic profiling is to learn patterns summarizing the given data, which are neither too specific nor too general (to be practically useful). Here, the dataset itself is a trivial overly specific profile, whereas the regex `.*` is an overly general one.

DEFINITION 3.1. Syntactic Profile: *Given a set of strings $\mathcal{S}$, syntactic profiling involves learning* ***(1)*** *a partitioning $\{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$ of the set $\mathcal{S}$, and* ***(2)*** *a set of syntactic patterns $\widetilde{P} = \{P_1, \ldots, P_k\}$, such that each P_i succinctly describes the partition $\mathcal{S}_i$. The set $\widetilde{P}$ is called a* syntactic profile *of $\mathcal{S}$.*

At a high level, our approach uses a clustering algorithm, which invokes pattern learning on demand. In Figure 2, we outline our PROFILE algorithm, which is parameterized by:

- a *pattern learner* $\mathscr{L}$ that accepts a set of strings $\mathcal{S}$, and returns a set of patterns consistent with them,
- a *cost function* that returns a numeric cost for a given pattern with respect to a given dataset

PROFILE accepts a dataset $\mathcal{S}$, with bounds $[\,m, M\,]$ for the number of patterns desired, a real number θ (a sampling factor), and returns a profile $\widetilde{P} = \{\langle\mathcal{S}_1, P_1\rangle, \ldots, \langle\mathcal{S}_k, P_k\rangle\}$ with $m \leqslant k \leqslant M$. First, it invokes BUILDHIERARCHY to construct a hierarchy H (line 1), which is subsequently partitioned by SPLIT (line 3). Finally, for each partition X, LEARNBESTPATTERN is invoked (line 4) to learn a pattern P, which is added to $\widetilde{P}$ paired with its partition X (line 5).

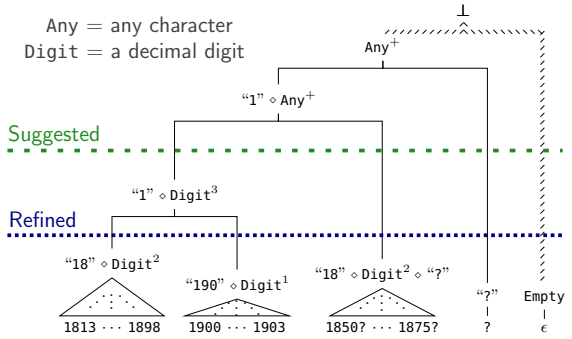
In the following subsections, we use Figure 1(a) as our running example and explain the two main components :

- BUILDHIERARCHY: for hierarchical clustering,
- LEARNBESTPATTERN: for pattern learning,

and their dependence on $\mathscr{L}$ and $\mathcal{C}$.

3.1 Hierarchical Clustering with Sampling

BUILDHIERARCHY yields a hierarchy (or *dendrogram*) over a given dataset $\mathcal{S}$, representing a nested grouping of strings based on their similarity [25]. For instance,



(a) An example hierarchy showing suggested and refined partitionings. Leaves indicate strings, and internal nodes indicate patterns matching them.

	ϵ	?	1817	1872	1901	1875?
ϵ	0	304.8	304.8	304.8	304.8	304.8
?	⊥	0	304.8	304.8	304.8	304.8
1817	⊥	Any$^+$	0	1.780	2.164	226.1
1872	⊥	Any$^+$	"18"⋄Digit$^{\times 2}$	0	2.164	173.8
1901	⊥	Any$^+$	"1" ⋄ Digit$^{\times 3}$	"1" ⋄ Digit$^{\times 3}$	0	268.4
1875?	⊥	Any$^+$	"18" ⋄ Any$^+$	"187" ⋄ Any$^+$	"1" ⋄Any$^+$	0

(b) Computed dissimilarity values (upper triangle), and corresponding learned patterns (lower triangle).

Figure 3: Hierarchy over our running example.

Figure 3(a) shows a hierarchy over our running example. Once constructed, a hierarchy may be split at an appropriate height to extract a partitioning of desired granularity. This also enables a natural form of refinement – supplying the desired number of clusters. However, FlashProfile also supports fully automatic profiling, using a simple heuristic to inspect the hierarchy and suggest a partitioning. Figure 3(a) shows a heuristically suggested split (with 4 clusters) and a finer split (with 5 clusters) resulting from a refinement request.

Flat clustering methods like k-means [34], or k-medoids [26], generate a single partitioning with fixed number of clusters. Moreover, they are equally expensive[4], since they require computation of all pairwise dissimilarities.

Agglomerative Hierarchical Clustering (AHC): We use a standard AHC algorithm, which initializes each string in its own cluster and repeatedly merges a pair of clusters till only a single cluster remains [25]. AHC requires: **(1)** a *dissimilarity measure* over strings, and **(2)** a *linkage criterion* over clusters. The linkage criterion extends a dissimilarity measure over strings to one over clusters. We use a classic *complete-linkage* [48] criterion for linking clusters. However, existing similarity measures [17] do not capture the desired syntactic dissimilarity over strings. To enable users to profile their datasets using custom atoms for their domains, the dissimilarity measure must necessarily be sensitive to the specific set of atoms that are allowed.

EXAMPLE 1. If hexadecimal digits are allowed, the string `f005ba11` is more similar to `5ca1ab1e` than `scalable`. But without it, `f005ba11` seems equally similar to either string (an alphanumeric atom describes either of them).

Syntactic Dissimilarity Measure: We observe that the desired dissimilarity measure is not a property of the strings themselves, but of the patterns describing them – specifically, the most suitable pattern amongst a potentially

[4] Time complexity of k-means is $O(kf|\mathcal{S}|)$, for k clusters over dataset $\mathcal{S}$ with f features defined on strings. However, in our case, dissimilarity w.r.t. other strings is the only feature, and k-means would run in $O(k|\mathcal{S}|^2)$.

Lower $[a-z]$	BinDigit $[01]$	Space $[\ ]$	HexDigit Digit | $[a-fA-F]$
Upper $[A-Z]$	Digit $[0-9]$	DotDash $[.-]$	AlphaDigitSpace Alpha | Digit | Space
Alpha Upper | Lower	AlphaDigit Alpha | Digit	Punct DotDash | $[,:?/]$	Base64 AlphaDigit | $[+\backslash=]$
TitleCaseWord Upper^1 Lower$^+$	AlphaDash Alpha | $[-]$	AlphaSpace Alpha | Space	Symbol Punct | $[@\#\$\%\&\cdots]$

Table 1: The default set of atoms in $\mathcal{L}_P$. The regular expressions for atoms are shown below their names.

large number of consistent patterns. Using the learner $\mathscr{L}$ and cost function $\mathcal{C}$, we define the dissimilarity measure η as the minimum cost incurred by any pattern for describing a given pair of strings. We evaluate η in § 6.1, and show its superiority over classic character-based similarity measures, and machine-learned regression models.

Figure 3(b) shows dissimilarities inferred by FlashProfile for a few pairs of strings from our running example. We also show the least-cost patterns describing them using the default atoms listed in Table 1. The void ($\perp$) indicates a pattern-learning failure that arises because no pattern in FlashProfile matches both empty and non-empty strings. We associate equal measures of dissimilarity with $\perp$ and $\texttt{Any}^+$.

Adaptive Sampling and Approximation: While η captures a high-quality syntactic similarity, it is expensive to compute. With it, each pairwise dissimilarity computation during AHC would require learning and scoring of patterns, making our technique too expensive for large real datasets.

We address this challenge by using a two-stage sampling technique. **(1)** At the top-level, FlashProfile employs a Sample−Profile−Filter cycle: we sample a small subset of the data, profile it, and filter out data that is explained by the profile learned so far. In § 6.2, we show that a small sample of strings is often sufficient to learn a general pattern describing them. **(2)** While profiling each sample, our BuildHierarchy algorithm adaptively samples a few pairwise dissimilarities, and approximates the rest. The key domain-specific observation that enables this is, computing the dissimilarity for a pair of strings gives us more than just a measure – we also learn a pattern. We *test* this pattern on other pairs, to learn their approximate dissimilarity. This is typically much faster than learning new patterns.

Example 2. The pattern "1" $\diamond$ $\texttt{Digit}^{\times 3}$ learned for the strings { 1817 , 1901 }, also describes { 1872 , 1901 }, and may be used to compute their dissimilarity, without pattern learning. Figure 3(b) shows other such recurring patterns.

Although the pattern "1" $\texttt{Any}^+$ learned for { 1901 , 1875? } also describes { 1872 , 1875? }, there exists another pattern "187" $\texttt{Any}^+$ which indicates a lower syntactic dissimilarity (due to a larger overlap) between them. Therefore, the dissimilarities (and equivalently, patterns) to be sampled, need to be chosen carefully for accurate approximations. We detail our adaptive sampling, and approximation algorithms in § 4.2 and § 4.3 respectively.

3.2 Pattern Learning via Program Synthesis

An important aspect of our clustering based approach to profiling, described in § 3.1, is its generality. It may leverage any learning technique which provides:

- a pattern learner $\mathscr{L}$ over strings, for:
  - computing dissimilarity, and approximations
  - exposing the profile to end users
- a cost function $\mathcal{C}$ over patterns, for:
  - identifying the most suitable pattern
  - computing a measure of dissimilarity

func LearnBestPattern$_{\langle \mathscr{L},\mathcal{C}\rangle}$($\mathcal{S}$: String[])
Result: The minimum cost pattern with its cost, learned for $\mathcal{S}$.
1. $V \leftarrow \mathscr{L}(\mathcal{S})$
2. **if** $V = \emptyset$ **then return** ⟨Pattern: $\perp$, Cost: $\mathcal{C}_{\text{MAX}}$⟩
3. $P \leftarrow \arg\min_{P \in V}\ \mathcal{C}(P, \mathcal{S})$
4. **return** ⟨Pattern: P, Cost: $\mathcal{C}(P, \mathcal{S})$⟩

Figure 4: An algorithm for learning the best pattern for a given set of strings $\mathcal{S}$, using a pattern learner $\mathscr{L}$ and cost function $\mathcal{C}$.

Figure 4 lists our LearnBestPattern algorithm for learning the most suitable pattern, and its cost. First, we compute a set of patterns V consistent with a dataset $\mathcal{S}$ by invoking $\mathscr{L}$ (line 1). If pattern learning fails (line 2), we return $\perp$, and a high default cost $\mathcal{C}_{\text{MAX}}$ (line 3). Otherwise, we invoke $\mathcal{C}$ to compute the pattern P that has the minimum cost with respect to the given dataset $\mathcal{S}$ (line 4).

A natural candidate for implementing $\mathscr{L}$, is *inductive program synthesis* [32], which generalizes a given set of examples to the desired program over a specified domain-specific language (DSL). FlashProfile learns patterns over the DSL $\mathcal{L}_P$ which is extensible by end-users. Using inductive synthesis over $\mathcal{L}_P$, we implement the pattern learner $\mathscr{L}_P$, and a cost function $\mathcal{C}_P$ which realize $\mathscr{L}$ and $\mathcal{C}$ respectively.

A DSL for Patterns: Our language $\mathcal{L}_P$ describes the space of possible *patterns* for strings. A pattern $P \in \mathcal{L}_P$ *describes* a string s, i.e. $P[\![s]\!]$ = True iff s satisfies every constraint imposed by P. Patterns are defined as arbitrary sequences of atomic patterns (*atoms*). We assume a default set of atoms (listed in Table 1) which may be extended with arbitrary pattern-matching logic. A pattern describes a string, if each successive atomic match succeeds, and they match the string in its entirety.

Example 3. The pattern "18" $\diamond$ $\texttt{Digit}^{\times 2}$ $\diamond$ "?" matches 1875?, but not 1872 since the atom "?" fails to find a match. Unlike regexes, patterns in $\mathcal{L}_P$ must match entire strings. For instance, "18" $\diamond$ $\texttt{Digit}^{\times 2}$ matches 1872, but not 1875? since the suffix ? is left unmatched.

Pattern Synthesis: The inductive synthesis problem for pattern learning is: given a set of strings $\mathcal{S}$, learn a pattern $P \in \mathcal{L}_P$ that describes every string in $\mathcal{S}$, i.e. $\forall\, s \in \mathcal{S}$: $P[\![s]\!]$ = True. Since P is a sequence of atoms, we can decompose the synthesis problem for matching P into smaller problems for matching individual atoms. However, a naïve approach of tokenizing each string to (exponentially many) sequences of atoms and computing their intersection is quite expensive. Instead, our learner $\mathscr{L}_P : 2^{\mathbb{S}} \rightarrow 2^{\mathcal{L}_P}$, finds the intersection in an incremental fashion, at the boundaries of atomic matches, by computing a set of *compatible atoms*.[5] Furthermore, $\mathscr{L}_P$ is sound and complete over $\mathcal{L}_P$ patterns, restricted to the user-specified set of atoms.

$\mathscr{L}_P$ is built using PROSE [4] (also called FlashMeta [40]), a state-of-the-art inductive synthesis framework that is being deployed industrially [41]. It performs a top-down walk over the specified DSL grammar, at each step reducing a given synthesis problem on the desired program to smaller synthesis problems on subexpressions of the program, based on the reduction logic specified by the DSL designer. We explain the details of $\mathscr{L}_P$ in § 5.2.

[5] We denote the set of all strings as $\mathbb{S}$, and power set of a set X as 2^X.

func BuildHierarchy$_{\langle \mathscr{L},\mathcal{C}\rangle}(\mathcal{S}$: String[], M: Int, θ: Double)
Result: A hierarchy over $\mathcal{S}$ assuming at most M clusters, and edge-sampling factor θ.

1 · $D \leftarrow$ SampleDissimilarities$_{\langle \mathscr{L},\mathcal{C}\rangle}(\mathcal{S}, \lfloor \theta\, M \rfloor)$
2 · $\mathcal{A} \leftarrow$ ApproxDMatrix$(\mathcal{S}, D)$
3 · $H \leftarrow$ AHC$(\mathcal{S}, \mathcal{A})$
4 · **return** H

Figure 5: Hierarchical clustering using sampling and approximations over pairwise dissimilarities.[6].

Cost of Patterns: Once a set of consistent patterns is identified, a variety of strategies may be employed to identify the most desirable pattern w.r.t the dataset. Our cost function, $\mathcal{C}_P : \mathcal{L}_P \times 2^{\mathbb{S}} \rightarrow \mathbb{R}$, quantifies the suitability of each pattern with respect to the given dataset. Intuitively, it decides a trade-off between two opposing factors:

Specificity: select a pattern that does not over-generalize
Simplicity: select a compact (interpretable) pattern

Example 4. The pair {`Male` , `Female`} are matched by the patterns `Upper` ⋄ `Lower`$^+$ and `Upper` ⋄ `HexDigit` ⋄ `Lower`$^+$. Although the latter is a more specific pattern, it is overly complex. On the other hand, the pattern `Alpha`$^+$ is simpler and easier to interpret by end users, but is overly general.

To this end, each atom in $\mathcal{L}_P$ has an empirically fixed *static cost*, and a dataset-driven *dynamic weight*. The final cost of a pattern is the weighted sum of the cost of atoms that appear in it. We describe our cost function $\mathcal{C}_P$ in § 5.3.

4. HIERARCHICAL CLUSTERING

In this section, we first explain our algorithm for building a hierarchy and learning a profile by sampling dissimilarities for only a few pairs of strings. In § 4.4, we then discuss how to combine profiles generated over several samples of a dataset to generate an overall profile. Henceforth, we use the term *pair* to denote a pair of strings.

The procedure BuildHierarchy, listed in Figure 5, constructs a hierarchy H over a given dataset $\mathcal{S}$, assuming at most M clusters to be extracted, using the sampling factor θ. First, SampleDissimilarities computes a hashtable D (line 1), which maps only $O(\theta M|\mathcal{S}|)$ pairs sampled from $\mathcal{S}$, to the dissimilarities and best patterns computed for them. We formally define this dissimilarity measure in § 4.1, and detail the SampleDissimilarities algorithm in § 4.2. D is then used by ApproxDMatrix to create matrix $\mathcal{A}$ containing all pairwise dissimilarities over $\mathcal{S}$, approximating wherever necessary (line 2). Finally, $\mathcal{A}$ is used by AHC to construct the hierarchy H, using the complete-linkage criterion $\widehat{\eta}$ (line 3). We explain AHC based on approximations, in § 4.3.

4.1 Syntactic Dissimilarity

We first formally define a syntactic dissimilarity measure over strings based on the patterns learned by $\mathscr{L}$ (specifically, the one with *least cost*), which describe them together.

Definition 4.1. Syntactic Dissimilarity: *The syntactic dissimilarity of identical strings is* 0. *For strings* $s_1 \neq s_2$, *we define the syntactic dissimilarity as the minimum cost incurred by any pattern to describe both:*

$$\eta(s_1, s_2) \stackrel{\text{def}}{=} \begin{cases} \mathcal{C}_{\text{MAX}} & \text{if } V = \emptyset \\ \min\limits_{P \in V} \mathcal{C}(P, \{s_1, s_2\}) & \text{otherwise} \end{cases}$$

where $V = \mathscr{L}(\{s_1, s_2\})$ *denotes the patterns learned for* $\{s_1, s_2\}$, *and* $\mathcal{C}_{\text{MAX}}$ *denotes a high cost for a learning failure.*

[6] $\lfloor x \rfloor$ denotes the *floor* of x, i.e. $\lfloor x \rfloor = \max\{m \in \mathbb{Z} \mid m \leqslant x\}$.

func SampleDissimilarities$_{\langle \mathscr{L},\mathcal{C}\rangle}(\mathcal{S}$: String[], $\widehat{M}$: Int)
Result: A dictionary mapping a few pairs of strings from $\mathcal{S}$, to the best pattern describing them and its cost.

1 · $a \leftarrow$ a random string in $\mathcal{S}$
2 · $D \leftarrow \emptyset$; $\rho \leftarrow \{a\}$
3 · **for** $i \leftarrow 1$ **to** $\widehat{M}$ **do**
4 · **for all** $b \in \mathcal{S}$ **do**
5 · $D[a, b] \leftarrow$ LearnBestPattern$_{\langle \mathscr{L},\mathcal{C}\rangle}(\{a, b\})$
► Pick the most dissimilar string w.r.t. strings in ρ.
6 · $a \leftarrow \arg\max_{x \in \mathcal{S}} \min_{y \in \rho} D[y, x]$.Cost
7 · ρ.add(a)
8 · **return** D

Figure 6: An algorithm for sampling patterns.

We detail FlashProfile's pattern learner $\mathscr{L}_P$ and cost function $\mathcal{C}_P$ in § 5, but it is easy to observe the relative syntactic dissimilarities for the following real-world entities based on patterns learned over atoms shown in Table 1.

Example 5. The dates shown below have a dissimilarity value of 1.64, based on the $\mathcal{L}_P$ pattern shown on the right:

2014-11-23
2014-02-09

“2014-” ⋄ `Digit`$^{\times 2}$ ⋄ “-” ⋄ `Digit`$^{\times 2}$

Whereas, the following dates in different formats have a noticeably higher dissimilarity value, 5.49:

2014-11-23
05/15/2010

`Digit`$^+$ ⋄ `Punct` ⋄ `Digit`$^{\times 2}$ ⋄ `Punct` ⋄ `Digit`$^+$

Finally, as one would expect, a date and an ISBN code have an extremely high degree of dissimilarity (198.4):

2014-11-23
978-3-642-28269-0

`Digit`$^+$ ⋄ “-” ⋄ `Digit`$^+$ ⋄ “-” ⋄ `Any`$^+$

We use the LearnBestPattern method (from Figure 4) for computing syntactic dissimilarity. LearnBestPattern returns the best (least-cost) pattern and its cost, for a given dataset $\mathcal{S}$. For computing $\eta(s_1, s_2)$, we provide $\mathcal{S} = \{s_1, s_2\}$.

4.2 Adaptive Sampling of Patterns

Although η accurately measures the syntactic dissimilarity of strings over an arbitrary language of patterns, it does so at the cost of performance. Using LearnBestPattern, every pairwise dissimilarity computation requires pattern learning and scoring, which may be computationally expensive depending on the number of consistent patterns generated. Moreover, typical clustering algorithms [25] require all pairwise dissimilarities. To put this into perspective, even with a fast pattern learner requiring ∼10 ms per dissimilarity computation[7], one would spend over 3 s on computing all 300 pairwise dissimilarities for profiling only 25 strings.

As shown in example 2, previously learned patterns may be used to estimate dissimilarities for other pairs. However, the patterns to be sampled must be chosen carefully. For instance, a pattern learned for a pair with very high dissimilarity may be too general, and may match many other pairs. But, approximations based on these matches would be inaccurate since the cost of this pattern may differ vastly from the *least-cost* patterns for other pairs.

Our adaptive sampling algorithm, shown in Figure 6, is inspired by the seeding technique of k-means++ [8]. SampleDissimilarities accepts a dataset $\mathcal{S}$, an integer $\widehat{M}$ that is provided by BuildHierarchy as $\lfloor \theta\, M \rfloor$ (line 1, Figure 5), and samples $O(\theta M|\mathcal{S}|)$ pairwise similarities,

[7] FlashProfile's pattern learner $\mathscr{L}_P$ has a median learning time of 7 ms per pairwise dissimilarity, over our benchmarks. For comparison, most recent synthesizers based on PROSE have a learning time of ∼500ms [41].

```
func ApproxDMatrix(S: String[],
                   D: String × String ↦ Pattern × Double)
Result: A complete dissimilarity matrix A over S.
1  A ← ∅
2  for all s ∈ S do
3      for all s′ ∈ S do
4          if s = s′ then A[s, s′] ← 0
5          else if ⟨s, s′⟩ ∈ D then A[s, s′] ← D[s, s′].Cost
6          else
▶              Approximate η(s, s′) with the best matching pattern in D.
7              X ← TestPatterns(D, {s, s′})
8              if X ≠ ∅ then A[s, s′] ← min_{d ∈ X} d.Cost
9              else
▶                  Compute η(s, s′) exactly if D has no matching pattern.
10                 D[s, s′] ← LearnBestPattern_⟨L,C⟩({s, s′})
11                 A[s, s′] ← D[s, s′].Cost
12 return A
```

Figure 7: An algorithm for generating a matrix of pairwise approx. dissimilarities over $\mathcal{S}$.

out of $O(|\mathcal{S}|^2)$. It iteratively selects a set ρ of $\widehat{M}$ strings from $\mathcal{S}$ that are most dissimilar to each other. The string that is most dissimilar to those existing in ρ, is the one that is the most dissimilar to its nearest neighbor in ρ (line 6). Instead of computing all pairwise dissimilarities over $\mathcal{S}$, SampleDissimilarities only computes the pairwise dissimilarities of strings in $\mathcal{S}$ w.r.t. those in ρ (line 5).

Example 6. Consider sampling dissimilarities over our running example from Figure 1(a) with $\widehat{M} = 4$, starting with `1817`. Based on the dissimilarities shown in Figure 3(b), we sample $\rho = \{\texttt{1817}, \epsilon, \texttt{?}, \texttt{1875?}\}$, in that order. With $\widehat{M} = 5$, we pick the next candidate for ρ as `1901`.

Intuitively, SampleDissimilarities samples a subset of dissimilarities that are likely to be sufficient for constructing a hierarchy accurate till M levels, i.e. if $1 \leqslant k \leqslant M$ clusters are extracted from the hierarchy, they should be equivalent to those extracted from a hierarchy constructed without any approximations. Since a hierarchy may be split to at most M clusters, θ should typically be at least 1.0 to sample intra- and inter- cluster dissimilarities for *at least* one point from each cluster. However, with expressive pattern languages, the sampled *least-cost* patterns may be too specific and may necessitate increasing θ to sample more patterns.

While a high θ degrades performance due to a large number of calls to the pattern learner, a low θ results in poor approximations, especially over large datasets. In §6.2 we show that $\theta = 1.25$ works well for real datasets.

4.3 Approximately Correct AHC

Our approximation technique is grounded on the observation that testing whether a pattern matches a string, is typically much faster than learning new patterns. If the dissimilarity of a pair is unknown, we test the sampled set of patterns, and approximate the dissimilarity as the minimum cost incurred by any known pattern that is consistent with the pair.

Using the hashtable D from SampleDissimilarities, the method ApproxDMatrix, listed in Figure 7, approximates a matrix $\mathcal{A}$ of all pairwise dissimilarities over $\mathcal{S}$. For identical strings, the dissimilarity is set to 0 (line 4). If the dissimilarity of a pair has already been sampled in D, we simply copy it to $\mathcal{A}$ (line 5). Lines 7, 8 show the key approximation step that are executed if the dissimilarity of a pair $\{s, s'\}$ has not been sampled. In line 7, we invoke TestPatterns to select a set X containing only those patterns P from D, which describe both s and s', i.e. $P[\![s]\!] = P[\![s']\!] = \texttt{True}$. If X is non-empty, we simply select the least-cost pattern from X (line 8). If X turns out to be empty, i.e. no sampled pattern describes both s and s', then we invoke LearnBestPattern to compute $\eta(s, s')$ (line 10). We also add the newly learned pattern to D, in line 11, to use it for subsequent approximations.

```
func AHC(S: String[], A: String × String ↦ Double)
Result: A hierarchy over S using dissimilarity matrix A.
1  H ← {MakeLeaf(s) | s ∈ S}
2  while |H| > 1 do
3      ⟨X, Y⟩ ← arg min_{X,Y ∈ H} η̂(X, Y | A)
4      H ← (H \ {X, Y}) ∪ {MakeNode(X, Y)}
5  return H
```

Figure 8: A standard AHC algorithm.

Once we have all pairwise dissimilarities, we use a standard AHC algorithm [25], shown in Figure 8. Initially in line 1, each string starts in its own cluster, which form the leaf nodes in the hierarchy. In each iteration, we select the least dissimilar pair of clusters (line 3) and join them to a new internal node (line 4), till we are left with a single node (line 2), which becomes the root of the hierarchy. To select the least dissimilar pair of clusters, AHC algorithms require a *linkage criterion* (line 3). We use a classic *complete-linkage* criterion [48] over our pairwise dissimilarities:

Definition 4.2. Complete-Linkage Criterion: *Given two clusters $X, Y \subseteq \mathcal{S}$ and a dissimilarity matrix $\mathcal{A}$ over $\mathcal{S}$, we define the overall dissimilarity of clusters X and Y as:*

$$\widehat{\eta}(X, Y \mid \mathcal{A}) \stackrel{\text{def}}{=} \max_{s_1 \in X,\, s_2 \in Y} \mathcal{A}[s_1\,,\, s_2]$$

where $\mathcal{A}[s_1\,,\, s_2]$ indicates the dissimilarity of s_1 and s_2.

Compared to other linkage criteria, complete linkage has been shown to be more resistant to outliers and yield useful hierarchies in many practical applications [25].

After a hierarchy H has been constructed, our Profile algorithm (listed in Figure 2) invokes the Split method to extract $m \leqslant k \leqslant M$ clusters. If $m = M$, it simply splits the first m levels (the top-most m internal nodes) of the hierarchy. Otherwise, it uses a heuristic based on the *elbow* (also called *knee*) method. Between the m^{th} and the M^{th} level, it locates a level k in the hierarchy after which the dissimilarities of subsequent levels do not seem to vary much.

4.4 Profiling Large Datasets

To scale our technique to large datasets, we implement a second round of sampling around our core Profile method. Recall that, the SampleDissimilarities samples $O(\theta M|\mathcal{S}|)$ pairwise dissimilarities by selecting a set ρ of most dissimilar strings, and computing their dissimilarity with all strings in dataset $\mathcal{S}$. But, although $|\rho| = \lfloor \theta M \rfloor$ is very small, $|\mathcal{S}|$ is still very large (several thousands) for real-life datasets.

Our second sampling technique runs the Profile algorithm from Figure 2 on small chunks of the original dataset and combines the generated profiles. We observe that even a small randomly selected subset exhibits relatively frequent patterns in the dataset, and our learner $\mathscr{L}_P$ does not require more than a couple of strings to learn them in most cases.

We implement a simple Sample−Profile−Filter loop:

1. sample $\lfloor \mu M \rfloor$ strings from $\mathcal{S}$,
2. add their profile to the current set of known patterns,
3. remove strings described by the known patterns from $\mathcal{S}$.

func CompressProfile$_{(\mathscr{L},\mathcal{C})}$($\widetilde{P}$: ref Profile, M: Int)
Result: An compressed profile $\widetilde{P}$ that satisfies $|\widetilde{P}| \leqslant M$.

1. **while** $|\widetilde{P}| > M$ **do**
▶ Compute the most similar partitions in the profile so far.
2. $\langle X, Y\rangle \leftarrow \arg\min_{X,Y \in \widetilde{P}}$ LearnBestPattern$_{(\mathscr{L},\mathcal{C})}(X.\text{Data} \cup Y.\text{Data}).\text{Cost}$
▶ Merge the partitions and update $\widetilde{P}$.
3. $Z \leftarrow X.\text{Data} \cup Y.\text{Data}$
4. $P \leftarrow$ LearnBestPattern$_{(\mathscr{L},\mathcal{C})}(Z).\text{Pattern}$
5. $\widetilde{P} \leftarrow (\widetilde{P} \setminus \{X, Y\}) \cup \{\langle \text{Data: } Z, \text{Pattern: } P\rangle\}$
6. **return** $\widetilde{P}$

Figure 9: An algorithm for compressing a profile.

The sampling factor μ determines the size of each chunk provided to Profile. A very small μ degrades performance, since in each iteration it learns a profile that does not generalize well over the remaining dataset, and thus requires many iterations to profile the entire dataset. A large μ also degrades performance, since it requires Profile to sample many pairwise dissimilarities. In §6.3, we show that $\mu = 4.0$ works well for FlashProfile, over most real-life datasets.

When adding new profiles learned in step (2), we may exceed the maximum number of patterns M specified by the user, and may need to *compress* the profile. Our CompressProfile algorithm, shown in Figure 9, accepts a large profile $\widetilde{P}$ and modifies it to have at most M patterns. $\widetilde{P}$ must be of the same type as returned by Profile, i.e. must contains patterns paired with corresponding data partitions. First, we identify a pair of partitions $\langle X, Y\rangle$ which are the most similar, i.e. incur the least cost for describing them together with a pattern (line 1). We then merge the data in X, Y to a single partition Z (line 3), learn a pattern describing Z (line 4), and update the profile $\widetilde{P}$ by replacing X, Y with Z and its pattern (line 5). CompressProfile repeats this entire process till the total number of patterns falls within the upper-bound M.

5. PATTERN SYNTHESIS

In this section we describe FlashProfile, which instantiates the proposed profiling technique based on clustering. We begin with a brief description of the language in § 5.1. In § 5.2, we present our pattern learner $\mathscr{L}_P$, which produces *all* patterns consistent with a given dataset using a user-specified set of atoms over $\mathcal{L}_P$. Finally, in § 5.3, we provide a description of our cost function $\mathcal{C}_P$ for $\mathcal{L}_P$ patterns.

5.1 The Pattern Language $\mathcal{L}_P$

Figure 10(a) shows formal syntax for our pattern language $\mathcal{L}_P$. Each pattern $P \in \mathcal{L}_P$ is a function P: String $\to$ Bool which embodies a set of constraints over strings. A pattern P *describes* a given string s i.e. $P[\![s]\!] =$ True, iff s satisfies all constraints imposed by P. Patterns in $\mathcal{L}_P$ are defined in terms of atomic patterns:

Definition 5.1. Atomic Patterns (Atoms): *An atom, α: String $\to$ Int is an operator which given a string s, returns the length of the longest prefix of s matched by it. Atoms only match non-empty strings, i.e. $\alpha(s) = 0$ indicates match failure for α on the string s.*

$\mathcal{L}_P$ allows the following four kinds of atoms:

- *Constant String*: $\texttt{Const}_s$ only matches the string s. We use a string literal such as "data" to denote $\texttt{Const}_{\text{"data"}}$.
- *Regular Expressions*: $\texttt{RegEx}_r$ matches the longest prefix matched by the regular expression r.

$$\begin{array}{lll} \text{Pattern } P[s] & := \texttt{Empty}(s) & \\ & \mid P[\texttt{SuffixAfter}(s, \alpha)] & \\ \text{Atom } \alpha & := \texttt{Class}_c^z \mid \texttt{RegEx}_r & \\ & \mid \texttt{Funct}_f \mid \texttt{Const}_s & \end{array} \qquad \begin{array}{l} c \in \text{sets of characters} \\ f \in \text{functions String} \to \text{Int} \\ r \in \text{regular expressions} \\ s \in \text{strings } \mathbb{S} \\ z \in \text{non-negative integers} \end{array}$$

(a) Syntax of an $\mathcal{L}_P$ pattern P.

$$\frac{s = \epsilon}{\texttt{Empty}(s) \;\Downarrow\; \text{true}} \qquad \frac{s = s_0 \circ s_1 \qquad \alpha(s) = |s_0| > 0}{\texttt{SuffixAfter}(s, \alpha) \;\Downarrow\; s_1}$$

$$\frac{}{\texttt{Funct}_f(s) \;\Downarrow\; f(s)} \qquad \frac{s' = s \circ s''}{\texttt{Const}_s(s') \;\Downarrow\; |s|}$$

$$\frac{L = \{x \mid \forall n \in \mathbb{N}, r \triangleright s[0:n]\}}{\texttt{RegEx}_r(s) \;\Downarrow\; \max_{x \in L} |x|}$$

$$\frac{s = s_0 \circ s_1 \qquad \forall x \in s_0 : x \in c \qquad s[|s_0|] \notin c}{\texttt{Class}_c^0(s) \;\Downarrow\; |s_0|}$$

$$\frac{s = s_0 \circ s_1 \qquad \forall x \in s_0 : x \in c \qquad |s_0| = z > 0 \qquad s[|s_0|] \notin c}{\texttt{Class}_c^z(s) \;\Downarrow\; z}$$

(b) Semantics of $\mathcal{L}_P$ patterns. The judgement $E \Downarrow v$ denotes that the expression E evaluates to a value v.

Figure 10: DSL $\mathcal{L}_P$ for defining syntactic patterns[8].

- *Functions*: $\texttt{Funct}_f$ allows arbitrary logic in the function f, and matches a prefix of length $|f(s)|$.
- *Character Class*: $\texttt{Class}_c^z$ matches the longest prefix that only contains characters from the set c, for $z = 0$. However, a *width* $z > 0$ enforces a fixed-width constraint as well – the match $\texttt{Class}_c^z(s)$ fails if $\texttt{Class}_c^0(s) \neq z$, otherwise it returns z.

FlashProfile provides a default set of atoms listed in Table 1. However, the set of atoms may be extended by end users to generate rich profiles for their datasets.

Example 7. Atom Digit is $\texttt{Class}_D^1$ with $D = \{0, \ldots, 9\}$. $\texttt{Digit}^+$ is $\texttt{Class}_D^0$, and $\texttt{Digit}^{\times 2}$ is $\texttt{Class}_D^2$. The atom $\texttt{Digit}^{\times 2}$ matches the string 04/23/2017 but not 2017/04/23. For the latter case, the longest prefix matched by $\texttt{Digit}^+$ is 2017, of length $4 \neq 2$. However, $\texttt{Digit}^+$ matches both strings, and returns 2 and 4 respectively.

Definition 5.2. Patterns: *A pattern is defined by a sequence of zero or more atoms. The $\mathcal{L}_P$ expression* Empty *denotes an empty sequence of atoms, which only matches the empty string ϵ. The pattern $\alpha_1 \diamond \alpha_2 \diamond \ldots \diamond \alpha_k$ is denoted by*

$$\texttt{Empty(SuffixAfter(}\cdots\texttt{SuffixAfter}(s, \alpha_1)\cdots, \alpha_k\texttt{))}$$

in $\mathcal{L}_P$, which matches a string s_0, iff

$$s_k = \epsilon \;\wedge\; \forall i \in \mathbb{N}, \alpha_i(s_{i-1}) > 0$$

where $s_{i+1} = s_i[\alpha_{i+1}(s_i) :]$ i.e. the remaining unmatched suffix of string s_i after matching atom α_{i+1}.

Formal semantics for atoms and patterns are shown in Figure 10(b). $s[i]$ denotes the i^{th} character of s, and $s[i:j]$ denotes the substring of s from the i^{th} character, till before the j^{th} character. j may be omitted to indicate a substring extending till the end of s. In $\mathcal{L}_P$, the $\texttt{SuffixAfter}(s, \alpha)$ operator computes $s[\alpha(s) :]$, or raises an error if $\alpha(s) = 0$.

Example 8. Consider the following URLs collected from a dataset* containing flight data for various destinations:

```
http://www.jetradar.com/flights/EsaAla-ESA/
http://www.jetradar.com/flights/Mumbai-BOM/
http://www.jetradar.com/flights/NDjamena-NDJ/
http://www.jetradar.com/flights/Bangalore-BLR/
http://www.jetradar.com/flights/LaForges-YLF/
```

The following pattern describes these URLs:

$$\text{"http://www.jetradar.com/flights/"} \diamond \texttt{Upper}^+ \diamond \texttt{Alpha}^+ \diamond \text{"-"} \diamond \texttt{Upper}^{\times 3} \diamond \text{"/"}$$

[8] $r \triangleright x$ denotes a successful match of regex r with string x in its entirety, and $a \circ b$ denotes the concatenation of strings a and b.

* https://support.travelpayouts.com/hc/ru/article_attachments/201368927/places_t.csv

5.2 Synthesis of $\mathcal{L}_P$ Patterns

FlashProfile's pattern learner $\mathscr{L}_P$ uses inductive program synthesis [32] for synthesizing patterns that describe a given set $\mathcal{S}$ of strings using a user-specified set of atoms $\mathcal{A}$. For convenience of end users, we automatically *enrich* their specified set of atoms by allowing: **(1)** all `Const` atoms, and **(2)** fixed-width variants of all `Class` atoms specified by them, for all widths. $\mathscr{L}_P$ is instantiated with these enriched atoms derived from $\mathcal{A}$, which we denote as $\langle\mathcal{A}\rangle$:

$$\begin{aligned}\langle\mathcal{A}\rangle = \mathcal{A} \quad &\cup \quad \{\texttt{Const}_s \mid s \in \mathbb{S}\} \\ &\cup \quad \{\texttt{Class}_c^z \mid \texttt{Class}_c^0 \in \mathcal{A} \wedge z \in \mathbb{N}\}\end{aligned} \tag{1}$$

Although $\langle\mathcal{A}\rangle$ is unbounded (since $\mathbb{S}$ is unbounded), as we explain later, our synthesis procedure only explores a small fraction of $\langle\mathcal{A}\rangle$ which are *compatible* with a given dataset $\mathcal{S}$.

We build on a state-of-the-art inductive program synthesis library PROSE [4], which implements the FlashMeta [40] framework. Our synthesis relies on *deductive reasoning* – reducing a synthesis problem over an expression, to smaller synthesis problems over its subexpressions. PROSE provides a convenient framework with highly efficient algorithms and data-structures for building such program synthesizers.

An inductive program synthesis task is defined by: **(1)** a domain-specific language (DSL) for target programs which in our case is $\mathcal{L}_P$, **(2)** a *spec* [40, §3.2] that defines a set of constraints over the output of the desired program. The spec given to $\mathscr{L}_P$ (denoted as φ), simply requires the desired pattern P to describe all strings i.e. $\forall s \in \mathcal{S}\colon P[\![s]\!] = \texttt{True}$. We formally write such a spec φ as:

$$\varphi \stackrel{\text{def}}{=} \textstyle\bigwedge_{s\,\in\,\mathcal{S}} [s \rightsquigarrow \texttt{True}]$$

Deductive reasoning allows us to reduce the spec φ over a pattern P to specs over its arguments. The specs are reduced recursively till terminal symbols in $\mathcal{L}_P$ (string s, or atom α). Then, starting from the values of terminal symbols which satisfy their spec, we collect the subexpressions and combine them to synthesize bigger expressions. We refer the reader to [40] for more details on the synthesis process.

The reduction logic (called *witness functions* [40, §5.2]) for specs is domain-specific, and depends on the semantics of the DSL operators. Specifically for $\mathcal{L}_P$, we need to define the logic for reducing the spec φ over the two kinds of patterns: `Empty`, and $P\,[\texttt{SuffixAfter}(s, \alpha)]$.

For `Empty`(s) to satisfy φ, i.e. describe all strings $s \in \mathcal{S}$, each string s must indeed be ϵ. No further reduction is needed since s is a terminal. We simply check, $\forall s \in \mathcal{S}\colon s = \epsilon$. `Empty`$(s)$ fails to satisfy φ if $\mathcal{S}$ contains non-empty strings.

$P\,[\texttt{SuffixAfter}(s, \alpha)]$ allows more complex patterns that first match s with atom α, and the remaining suffix with P. In contrast to prior approaches [43, 45] which learn complete patterns over individual strings and then combine them, in $\mathscr{L}_P$ we compute an intersection of patterns consistent with individual strings, in an incremental manner. In pattern $P\,[\texttt{SuffixAfter}(s, \alpha)]$, the unknowns are α (the atom which matches a prefix of s), and P (the pattern which matches the remaining suffix $s[\alpha(s)\texttt{:}]$). However, for a fixed α, P can be synthesized recursively, by posing a synthesis problem similar to the original one, over the suffix $s[\alpha(s)\texttt{:}]$ instead of s, for each $s \in \mathcal{S}$ i.e. with the new spec:

$$\varphi_\alpha \stackrel{\text{def}}{=} \textstyle\bigwedge_{s\,\in\,\mathcal{S}} \left[s[\alpha(s)\texttt{:}] \rightsquigarrow \texttt{True}\right] \tag{2}$$

Instead of enumerating φ_α for all allowed atoms, we consider only those atoms α that succeed in matching some prefix of *all* strings in $\mathcal{S}$. This is the key to computing the intersection of patterns over individual strings in an incremental fashion. Such atoms are said to be *compatible* with the given dataset $\mathcal{S}$.

```
func GetMaxCompatibleAtoms(S: String[], A: Atom[])
Result: The maximal set of atoms that are compatible with S.
1  · C ← ∅ ; Λ ← A
2  · for all s ∈ S do
3  ·     for all α ∈ Λ do
   ►         Remove incompatible atoms.
4  ·         if α(s) = 0 then
5  ·             Λ.Remove(α) ; C.Remove(α)
6  ·         else if α ∈ Class then
7  ·             if α ∉ C then C[α] ← α(s)
8  ·             else if C[α] ≠ α(s) then C.Remove(α)
   ► Add compatible fixed-width Class atoms.
9  · for all α ∈ C do
10 ·     Λ.Add(RestrictWidth(α, C[α]))
   ► Add compatible Const atoms.
11 · L ← LongestCommonPrefix(S)
12 · Λ.Add(Const_L[0:1], Const_L[0:2], ..., Const_L)
13 · return Λ
```

Figure 11: Algorithm for computing $\max_\propto^{\langle\mathcal{A}\rangle}[\mathcal{S}]$ on a dataset $\mathcal{S}$, with a user-specified set of atoms $\mathcal{A}$.

Definition 5.3. Compatible Atoms: *Over a set U of allowed atoms, we say a set of atoms $A \subseteq U$ is compatible to a given dataset $\mathcal{S}$, denoted as $A \propto \mathcal{S}$, if every atom in A successfully matches some prefix of every string $s \in \mathcal{S}$, i.e.*

$$A \propto \mathcal{S} \quad \text{iff} \quad \forall\,\alpha \in A\colon \forall\, s \in \mathcal{S}\colon\ \alpha(s) > 0.$$

We call a compatible set A of atoms is maximal *under U, denoted as $A = \max_\propto^U[\mathcal{S}]$ iff $\forall\, X \subseteq U\colon\ X \propto \mathcal{S} \Rightarrow X \subseteq A$.*

Example 9. Consider the following Canadian postal codes: $\mathcal{S} = \{$ `V6E3V6`, `V6C2S6`, `V6V1X5`, `V6X3S4` $\}$. Over the set $\langle\mathcal{A}\rangle$ of atoms using equation (1) on $\mathcal{A}$ = the default atoms (listed in Table 1), the maximal set of compatible atoms $\max_\propto^{\langle\mathcal{A}\rangle}[\mathcal{S}]$ contains 18 atoms such as "`V6`", "`V`", `Upper`, `Upper`$^+$, `Alpha`$^+$, `AlphaSpace`, `AlphaDigit`$^{\times 6}$ etc.

GetMaxCompatibleAtoms, outlined in Figure 11, accepts a set of strings $\mathcal{S}$, a set of atoms $\mathcal{A}$ specified by the user, and computes the set $\Lambda = \max_\propto^{\langle\mathcal{A}\rangle}[\mathcal{S}]$, where $\langle\mathcal{A}\rangle$ denotes the enriched set of atoms based on $\mathcal{A}$ given by equation (1). We start with $\Lambda = \mathcal{A}$ (line 1), and gradually remove atoms that are not compatible with $\mathcal{S}$, i.e. fail to match at least one string $s \in \mathcal{S}$ (line 5). For enriching $\mathcal{A}$ with fixed-width `Class` tokens, we maintain a hashtable C that maps a `Class` token to its width (line 8). If the width of a `Class` atom is not constant over all $s \in \mathcal{S}$, we remove it from C (line 9). We finally add to Λ, the fixed-width variants of all `Class` atoms in C, generated by calling RestrictWidth (line 10). For enriching with `Const` atoms, we compute the longest common prefix L across all $s \in \mathcal{S}$ (line 11), and add every prefix of it to Λ, as a compatible `Const` token (line 12).

In essence, a spec φ for $P\,[\texttt{SuffixAfter}(s, \alpha)]$ is reduced to a set of $\left|\max_\propto^{\langle\mathcal{A}\rangle}[\mathcal{S}]\right|$ new specs, each representing a distinct synthesis problem for P: $\{\varphi_\alpha \mid \alpha \in \max_\propto^{\langle\mathcal{A}\rangle}[\mathcal{S}]\}$, where φ_α is as given by equation (2), and $\langle\mathcal{A}\rangle$ denotes the enriched set of atoms derived from $\mathcal{A}$ by equation (1). Each synthesis problem is solved the same way as the original one. PROSE handles the recursive propagation of appropriate specs to subexpressions, and combines the generated subexpressions to patterns that satisfy the original spec φ.

We conclude with a comment on the soundness and completeness of our pattern learner $\mathscr{L}_P$.

Definition 5.4. Soundness and U-Completeness: *We say that a learner $\mathscr{L}_P$ for $\mathcal{L}_P$ patterns is* sound, *if for a given dataset $\mathcal{S}$ every learned pattern P satisfies $\forall s \in \mathcal{S} : P[\![s]\!] =$ `True`. We say that $\mathscr{L}_P$ instantiated with a set U of atoms is U-*complete, *if for every dataset $\mathcal{S}$, it learns all patterns $P \in \mathcal{L}_P$ over U atoms, which satisfy $\forall s \in \mathcal{S} : P[\![s]\!] =$ `True`.*

For our synthesis procedure, soundness is guaranteed since we only consider compatible atoms. Completeness follows from the fact that we always consider their maximal set over U atoms. $\mathscr{L}_P$ is $\langle\mathcal{A}\rangle$-complete for any user-specified set $\mathcal{A}$ of atoms. Therefore, once the set of patterns $\mathscr{L}_P(\mathcal{S})$ has been learned for a $\mathcal{S}$, a variety of cost functions may be employed to select the most suitable pattern for $\mathcal{S}$ amongst all possible patterns over $\langle\mathcal{A}\rangle$, without recomputing $\mathscr{L}_P(\mathcal{S})$.

5.3 Cost of Patterns in $\mathcal{L}_P$

Our cost function $\mathcal{C}_P$ produces a real number, given a pattern $P \in \mathcal{L}_P$ and a dataset $\mathcal{S}$, based on the structure of P and its behaviour over $\mathcal{S}$. `Empty` is assigned a cost of 0 regardless of the dataset, since any dataset with which `Empty` is consistent, only contain ϵ. We define the cost $\mathcal{C}_P(P, \mathcal{S})$ for a pattern $P = \alpha_1 \diamond \ldots \diamond \alpha_n$, with respect to dataset $\mathcal{S}$ as:

$$\mathcal{C}_P(P, \mathcal{S}) = \textstyle\sum_{i=1}^{n} \mathcal{C}_P(\alpha_i) \cdot D(i, P, \mathcal{S})$$

Each atom α in $\mathcal{L}_P$ has statically assigned cost $\mathcal{C}_P(\alpha) \in (0, \mathcal{C}_{\text{MAX}})$ based on a priori bias for the atom. The static costs for the default atoms in FlashProfile were empirically decided. We define $\mathcal{C}_{\text{MAX}} = \mathcal{C}_P(\texttt{Any}^+)$ to be the maximum cost across all atoms. For a pattern $P = \alpha_1 \diamond \ldots \diamond \alpha_n$, our cost function $\mathcal{C}_P$ sums these static costs after applying a dynamically determined weight $D(i, P, \mathcal{S}) \in (0, 1)$, based on how well each token α_i *generalizes* over $\mathcal{S}$.

$$D(i, P, \mathcal{S}) = \frac{1}{|\mathcal{S}|} \cdot \sum_{s_j \in \mathcal{S}} \frac{l_{j,i}}{|s_j|}$$

where $l_j = \langle l_{j,1}, \ldots, l_{j,n} \rangle$ denotes the lengths of successive prefix matches over a string $s_j \in \mathcal{S}$. Since a token match never fails over $\mathcal{S}$ for any synthesized pattern P, $l_{j,i} > 0$ and $D(i, P, \mathcal{S}) > 0$ for all tokens $\alpha_i \in P$.

Example 10. Consider $\mathcal{S} = \{\texttt{Male}, \texttt{Female}\}$, that are matched by $P_1 = \texttt{Upper} \diamond \texttt{Lower}^+$ and $P_2 = \texttt{Upper} \diamond \texttt{HexDigit} \diamond \texttt{Lower}^+$. The static costs for the relevant atoms are:

$$\{\texttt{Upper} \mapsto 8.2,\ \texttt{HexDigit} \mapsto 26.3,\ \texttt{Lower}^+ \mapsto 9.1\}$$

The costs for both patterns shown above, are computed as:

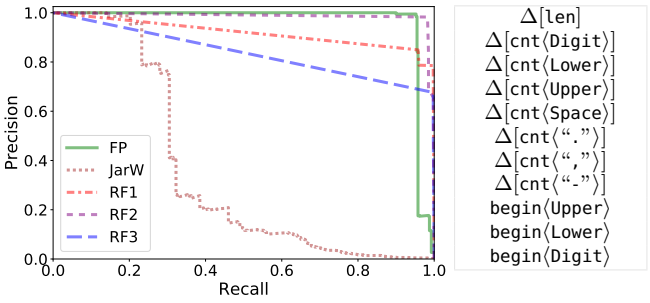
$$\mathcal{C}_P(P_1, \mathcal{S}) = \tfrac{8.2}{2}(\tfrac{1}{4} + \tfrac{1}{6}) + \tfrac{9.1}{2}(\tfrac{3}{4} + \tfrac{5}{6}) = 8.9$$

$$\mathcal{C}_P(P_2, \mathcal{S}) = \tfrac{8.2}{2}(\tfrac{1}{4} + \tfrac{1}{6}) + \tfrac{26.3}{2}(\tfrac{1}{4} + \tfrac{1}{6}) + \tfrac{9.1}{2}(\tfrac{2}{4} + \tfrac{4}{6}) = 12.5$$

P_1 is chosen as best pattern, since $\mathcal{C}_P(P_1, \mathcal{S}) < \mathcal{C}_P(P_2, \mathcal{S})$.

Note that although `HexDigit` is more specific compared to `Upper` and `Lower` – `HexDigit` contains 16 characters as opposed to 26, it has a higher static cost. This is an example of a priori bias against `HexDigit` to avoid strings like "`face`" being described as $\texttt{HexDigit}^{\times 4}$ instead of $\texttt{Lower}^{\times 4}$. However, its cost is much lower compared to $\mathcal{C}_P(\texttt{AlphaDigit}) = 639.6$, making it the preferred pattern for strings such as "`f00d`". Static costs are baked into the atoms and must be provided by domain experts when introducing new atoms.

$\mathcal{C}_P$ balances the trade-off between specificity vs generality – more specific atoms receive a smaller dynamic weight (which leads to a smaller overall cost), whereas the *sum* of the costs over many overly specific atoms may exceed the cost of a single more general atom. In § 6.2, we evaluate the quality of profiles learned by FlashProfile and show that they are *natural* – neither too specific, nor overly general.



(a) Precision-Recall curves **(b) RF Features**[10]

Measure	FP	JarW	RF_1	RF_2	RF_3
AUC	96.28%	35.52%	91.73%	98.71%	76.82%

Figure 12: Comparing the accuracy of FlashProfile (FP) with a character-based measure (JarW) and machine-learned models (RF_1, RF_2, RF_3). RF models show high variance w.r.t. training distribution.

6. EVALUATION

In this section, we present an experimental evaluation of FlashProfile, focusing on the following key questions:

- (§ 6.1) How well does our syntactic similarity measure perform over real world entities?
- (§ 6.2) How accurate are the profiles over real datasets, and what is the effect of sampling and approximations?
- (§ 6.3) What is the overall performance of FlashProfile, and how does it depend on the various parameters?
- (§ 6.4) Are the generated profiles natural and useful? How do they compare to those from existing tools?

Implementation and Experimental Setup: We have implemented FlashProfile as a cross-platform C# library built using Microsoft PROSE [4]. All experiments were performed on an 8-core Intel i7 3.60GHz machine with 32GB RAM running 64-bit Ubuntu 16.10 with .NET Core 1.0.1.

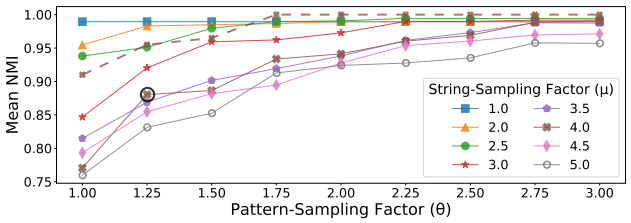
6.1 Syntactic Similarity

We evaluate the applicability of our dissimilarity measure from Definition 4.1, over real-life entities. From 25 clean real datasets[11] ranging over names, dates, postal codes, phone numbers etc. in different formats, we randomly pick two datasets, and select a random string from each. We picked 240400 such pairs of strings. A good similarity measure is expected to be able to identify when the two strings are drawn from the same dataset by assigning them a lower dissimilarity value, compared to two strings selected from different datasets. For example, the pair {`Albert Einstein` , `Isaac Newton`} should have a lower dissimilarity value than {`Albert Einstein` , `03/20/1998`}.

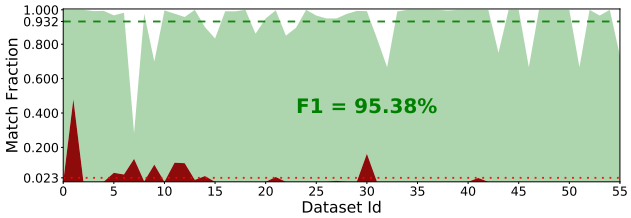
For evaluation, we use the standard precision-recall [35] (PR) measure. Precision in our context is the fraction of pairs that truly belongs to the same dataset, out of all pairs that are labeled to be "similar" by the predictor. Recall is the fraction of pairs retrieved by the predictor, out of all pairs truly drawn from same datasets. By varying the threshold for labelling a pair as "similar", we generate a PR curve and measure the area under the curve (called AUC). A good similarity measure should exhibit high precision and high recall, and therefore have a high AUC. Figure 12 show a comparison of our method against two

[10] `len` returns length of a string, `begin`$\langle X \rangle$ checks if a string begins with a character in X, `cnt`$\langle X \rangle$ counts occurences of characters from X in a string, and $\Delta[f]$ computes $|f(s_1) - f(s_2)|^2$ for a pair of strings s_1 and s_2.

[11] Available at: https://github.com/SaswatPadhi/ProfilingExperiments

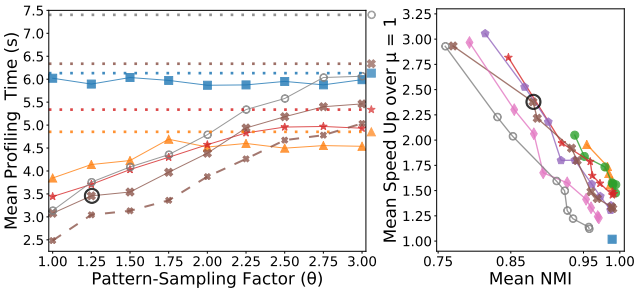


(a) Partitioning accuracy with different configurations.



(b) Quality of learned profiles with $\langle \mu = 4.0, \theta = 1.25 \rangle$.

Figure 13: Accuracy of FlashProfile.



(a) With different configurations (b) Perf ∼ Accuracy

Figure 14: Impact of sampling on performance. (using the same colors and markers as Figure 13(a))

baselines: **(1)** a character-based similarity measure (JarW), and **(2)** a machine-learned predictor (RF) using several intuitive syntactic features. We explain them below.

We observed that character-based measures [17] show poor AUC, and are not indicative of syntactic similarity. A popular data-wrangling tool OpenRefine [3] allows clustering of string data using Levenshtein distance [30]. However, this measure exhibits a negligible AUC over our benchmarks. Although the Jaro-Winkler distance [49] indicated as JarW in Figure 12(a) shows a better AUC, it is quite low compared to both our, and machine-learned predictors.

Our second baseline is a standard random forest [11] model RF using the syntactic features listed in Figure 12(b) such as, difference in length, number of digits etc. We train RF_1 over 160267 pairs with $\left(\frac{1}{25}\right)^2 = 0.16\%$ pairs drawn from same datasets. We observe from Figure 12(a) that the accuracy of RF is quite susceptible to changes in the distribution of the training data. RF_2 and RF_3 were trained with 0.64% and 1.28% pairs from same datasets, respectively. While RF_2 performs marginally better than our predictor, RF_3 performs worse. In contrast, our technique does not require any training, and hence even if training and test distributions are significantly different, our method still produces an accurate similarity measure.

6.2 Profiling Accuracy

We show FlashProfile's accuracy along two dimensions:

- *Partitions:* Our sampling and approximation techniques preserve accuracy of partitioning the datasets,
- *Descriptions:* Profiles generated using $\mathscr{L}_P$ and $\mathcal{C}_P$ are natural – neither too general nor too specific.

Partitioning: From 25 clean datasets, we randomly pick $n \in [2, 8]$ datasets, and select a group of 256 random strings from each. We combine them, and invoke FlashProfile to partition them into n clusters. We measure the precision of clustering using the *symmetric uncertainty* [50], which is a measure of normalized mutual information (NMI). An NMI of 1 indicates the resulting partitioning to be identical to the original groupings of strings, and an NMI of 0 indicates that the final partitioning is unrelated to the original groupings.

Recall that in each iteration, we profile $\lfloor \mu M \rfloor$ strings, and sample pairwise dissimilarities of only $|\rho| = \lfloor \theta M \rfloor$ strings w.r.t. $\mathcal{S}$. For this experiment, we set $M = n$. With different values of μ and θ, we show the mean NMI of the partitionings over 10 tests for each value of n, in Figure 13(a).

The NMI improves with increasing θ, since we sample more dissimilarities, resulting in better approximations. However, the NMI drops with increasing μ, since more pairwise dissimilarities are approximated. We observe that the median NMI is significantly higher than the mean, indicating a small number of cases where FlashProfile made poor approximations. The dashed line indicates the median NMIs with $\mu = 4.0$. We observe a median NMI of 0.96 (mean 0.88) for $\langle \mu = 4.0, \theta = 1.25 \rangle$, which is FlashProfile's default configuration for all our experiments (indicated by the circled point in Figure 13(a)).

Descriptions: We evaluate the suitability of the automatically suggested profiles, by measuring their precision and recall. A natural profile should not be overly specific – it should generalize well over the dataset (true positives), but not beyond it (false positives).

We consider 56 datasets ignoring datasets with duplicate formats. For each dataset, we profile a randomly selected subset containing 10% of its strings, and measure: **(1)** the fraction of the remaining dataset described by it, and **(2)** the fraction of an equal number of strings from other datasets, matched by it. Figure 13(b) summarizes our results. The lighter and darker shades indicate the fraction of true positives and false positives respectively. The white area at the top indicates the fraction of false negatives – the fraction of the remaining dataset that is not described by the profile. We record an overall precision of 97.8%, a recall of 93.4%. The dashed line indicates a mean true positive rate of 93.2%, and the dotted line shows a mean false positive rate of 2.3%.

6.3 Performance

We evaluate FlashProfile's performance over various $\langle \mu, \theta \rangle$-configurations considered during the partitioning-accuracy evaluation. We show the performance-accuracy trade off in Figure 14(b). We plot the mean speed up of various configurations over $\langle \mu = 1.0, \theta = 1.0 \rangle$, against the mean NMI of partitioning. Our default configuration $\langle \mu = 4.0, \theta = 1.25 \rangle$ achieves a mean speed up of 2.3×. We also show the profiling times in Figure 14(a). The dotted lines indicate profiling time without pattern sampling, for different values of the string-sampling factor μ. The dashed line shows the median profiling time for various values of θ with $\mu = 4$.

As one would expect, the profiling time increases with θ, since FlashProfile samples more patterns making more calls to $\mathscr{L}_P$. The dependence of profiling time on μ however, is more interesting. Notice that for $\mu = 1$, the profiling time is higher than any other configuration

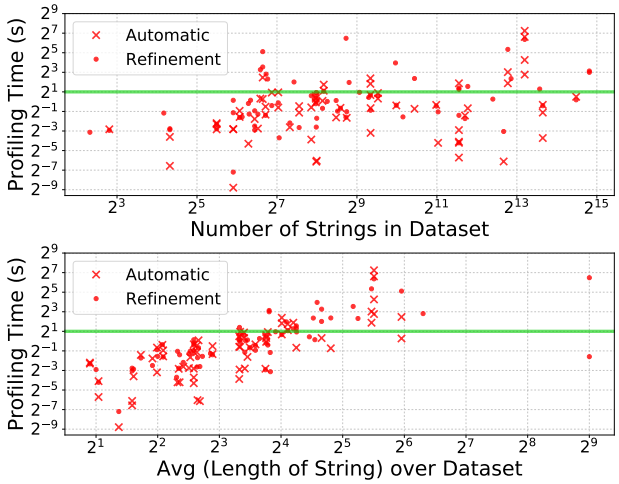


Figure 15: Profiling time for real-life datasets.

with pattern sampling enabled (solid lines). This is due to the fact that FlashProfile learns very specific profiles with $\mu = 1$ over a very small sample of strings, and does not describe much of the remaining data. This results in many Sample−Profile−Filter iterations over the entire dataset. The performance improves with μ till $\mu = 4.0$, and then starts deteriorating as we sample many more pairwise dissimilarities.

Finally, we evaluate FlashProfile's performance on end-to-end real-life profiling tasks on 74 datasets collected from various online sources, that have a mixture of clean and dirty datasets. Over 142 tasks – 74 for automatic profiling, and 68 for refinement, we observe a median profiling time of 0.7s. With our default configuration, 77% of the requests are fulfilled within 2 seconds – 73% of automatic profiling tasks, and 82% of refinement tasks. In Figure 15 we show the variance of profiling times w.r.t. size of the datasets (number of strings in them), and the average length of the strings in the datasets (all axes being logarithmic).

6.4 Comparison of Learned Profiles

For three real-life datasets, we compare the profiles learned by FlashProfile to patterns generated by three state-of-the-art tools: **(1)** Microsoft's SQL Server Data Tools [2] (SSDT), **(2)** Ataccama One [1] (A1) – a dedicated profiling tool, and **(3)** Potter's Wheel [43] (PW). In SSDT, we use `PercentageDataCoverageDesired = 100`, and the default limit (10) on the number of patterns.

Below, we list the automatically suggested profile from FlashProfile as FP, and the one generated on requesting exactly k patterns as FP_k. For brevity, we **(1)** omit $\diamond$, **(2)** denote a constant atom "data" as data, **(3)** abbreviate the default atoms: Digit $\mapsto$ d, Upper $\mapsto$ u, Space $\mapsto$ ␣, and **(4)** expand repeated atoms such as $\text{Digit}^{\times 3}$ to ddd.

Example 11. *A dataset containing postal codes*

Data: 99518, 61021-9150, 2645, K0K 2C0, 61604-5004,...
PW: Most frequent pattern = int

FP: ϵ | S7K7K9 | 61ddd-dddd | udu␣dud | d^+
A1: N | N-N | LDLDLD | LDL DLD
SSDT: \w\w\w \w\w\w | \d\d\d\d\d | \d\d\d\d | .*
FP_6: ϵ | S7K7K9 | 61ddd-dddd | udu␣dud | ddddd | dddd

SSDT produces an overly general profile containing .*. Ataccama One uses a restricted set of patterns that allow digits (D), numbers (N), letters (L), words (W). Moreover, it does not learn constants or fixed-width constraints.

Example 12. *A dataset containing U.S. routes*
(We abbreviate: AlphaSpace $\mapsto \omega$, AlphaDigitSpace $\mapsto \sigma$)

Data: OR-213, I-5 N, I-405 S, OR-99E, US-26 E, I-84 E,...
PW: Most frequent pattern = IspellWord int space AllCapsWord

Initially, FlashProfile suggests a conservative profile:

FP: ϵ | 12348 N CENTER | US 26(ω^+) | u^+-σ^+

Users may interactively refine the profiles. Even at the same level of granularity, for example with both A1 and FP generating 6 partitions below, FP qualifies them with richer patterns identifying relevant constants for the partition:

FP_7: ϵ | 12348 N CENTER | US 26(SUNSET) | US 26(MT HOOD HWY) | u^+-d^+ | uu-ddu$^+$ | u^+-d^+␣u^+
A1: N L W | W N (W) | W N (W W W) | W-N | W-NW | W-N W

Although SSDT identifies some constants, it merges many small but syntactically dissimilar partitions, producing .*:

FP_9: ϵ | 12348 N CENTER | US 26(SUNSET) | US 26(MT HOOD HWY) | u^+-d^+ | US-30BY | OR-99u | I-d^+␣u^+ | uu-2d^+␣u
SSDT: US-26 E | US-26 W | I-5 N | I-5 S | I-84 E | I-84 W | I-\d\d\d N | I-\d\d\d S | .*

We conclude with an interesting observation: requesting for 13 partitions in this case separates all the route prefixes:

FP_{13}: ϵ | 12348 N CENTER | US 26(SUNSET) | US 26(MT HOOD HWY) | I-5 | US-30 | OR-d^+ | US-30BY | OR-99u | I-5␣u^+ | I-d^+␣u | US-26␣u | OR-217␣u

7. APPLICATIONS IN PBE SYSTEMS

In this section, we briefly discuss applications of our data profiling technique to improve PBE systems. Such systems aim to synthesize a desired program from a small number of input-output examples [18, 21, 45]. For instance, given an example "Albert Einstein" $\rightsquigarrow$ "A.E.", the goal is to learn a program that extracts the initials for a given name. That is, given "Alan Turing", we want the synthesized program to output "A.T.". Though several PBE systems have been proposed recently, a major criticism for these systems has been the lack of usability and confidence in them [28, 37].

Examples are an inherently under-constrained form of specs for the desired program, and a large number of programs (up to 10^{20}) may be consistent with them [41]. Two major challenges to learning the desired program are: **(1)** obtaining a representative set of examples that convey the desired behavior, and **(2)** ranking the consistent programs to select the ones *natural* to end users.

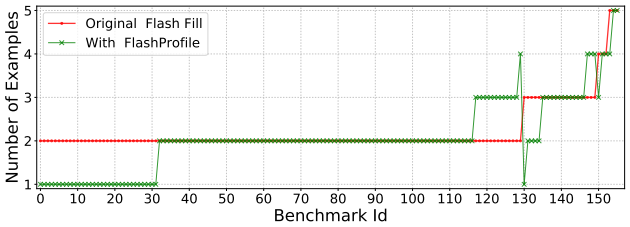
Significant Inputs: A user of the PBE system cannot be expected to provide the representative examples. In fact, typically, users provide outputs for only the first few inputs. However, if all these examples are very similar, the system may not learn a program that generalizes over other inputs. In § 2, we discuss such a case for Flash Fill [18], a popular PBE system for string transformations. Instead, we propose to request the user to provide outputs for *significant* inputs – the most dissimilar input w.r.t. those previously provided.

First, we generate a syntactic profile $\widetilde{P}$ for the input, and use the OrderPartitions function, listed in Figure 16(a), to order the input partitions, based on mutual dissimilarity. Starting with the tightest partition (line 1) i.e. the one with the least-cost pattern describing it, we iteratively append

func OrderPartitions$_{(\mathscr{L},\mathcal{C})}(\widetilde{P}$: Profile)
Result: An ordered list of partitions $\langle \mathcal{S}_1, \ldots, \mathcal{S}_{|\widetilde{P}|} \rangle$ of the source dataset $\mathcal{S}$.

▶ First select the least-cost partition.
1· $\rho \leftarrow \langle (\arg\min_{X \in \widetilde{P}} \mathcal{C}(X.\text{Pattern}, X.\text{Data})).\text{Data} \rangle$
2· **while** $|\rho| < |\widetilde{P}|$ **do**
▶ Select the partition that is most dissimilar w.r.t. existing partitions in ρ.
3· $T \leftarrow \arg\max_{Z \in \widetilde{P}} \min_{X \in \rho}$ LearnBestPattern$_{(\mathscr{L},\mathcal{C})}(Z.\text{Data} \cup X).\text{Cost}$
4· ρ.Append(T.Data)
5· **return** ρ

(a) Ordering partitions by mutual dissimilarity.



(b) Examples needed with and without significant inputs.

Figure 16: Significant inputs for Flash Fill.

the partition that is most dissimilar (requires the highest-cost pattern to describe) with prior partitions in ρ (line 3).

We request the user to provide an output for a randomly selected input from each partition in order $\langle \mathcal{S}_1, \ldots, \mathcal{S}_{|\widetilde{P}|} \rangle$. Since Flash Fill is interactive, the user can inspect the output for each input, and skip if it is already correct. Finally, we restart from $\mathcal{S}_1$ after one cycle through all partitions.

We measure the efficacy of our interaction model, over 163 Flash Fill benchmarks that require > 1 examples to learn the desired program. Figure 16(b) compares the number of examples required originally, to that using significant inputs. Seven benchmarks that timed-out have been omitted. Over the remaining 156 benchmarks, we observe that Flash Fill, **(1)** never requires a second example from the same partition, for 131 benchmarks, and **(2)** uses the smallest possible set of examples over the given inputs, for 140 benchmarks.

Thus, *(1)* validates our hypothesis that syntactic partitions indeed identify representative inputs, and *(2)* further indicates that ordering partitions using our dissimilarity measure, allows for a highly effective interaction model.

Note that, the significant inputs scenario is similar to *active learning*, which is well-studied in machine-learning literature [23]. Active learning also seeks to find data points to be annotated so that the learned predictor is most accurate. However, typical active-learning methods require hundreds of annotated examples. In contrast, PBE systems typically deal with very few annotated examples [37].

8. RELATED WORK

Data Profiling: There has been a line of work on profiling various aspects of a column of data; see [38, 6] for recent surveys. Traditional profiling techniques target simple statistical properties [36, 42, 12, 13, 24, 3, 5].

To our knowledge, no existing technique supports refinement of syntactic profiles learned over an extensible language. We present a novel dissimilarity measure which is the key to learning refinable profiles over user-specified patterns. While Potter's Wheel [43] does not learn a compile profile, it learns the most frequent data pattern using user-defined *domains*. SSDT [2] learns rich regular expressions but is neither extensible not comprehensive. A dedicated profiling tool Ataccama One [1] generates comprehensive profiles over a very small set of base patterns. OpenRefine [3] does not learn syntactic profiles, but it allows clustering of strings using character-based similarity measures [17]. In §6 we show that they do not capture syntactic similarity.

Application-Specific Structure Learning: There has been prior work on learning specific structural properties aimed at aiding data wrangling applications, such as data transformations [43, 45], information extraction [31], and reformatting or normalization [44, 27]. These approaches make specific assumptions regarding the target application, which do not necessarily hold when learning general purpose comprehensive profiles for data understanding. We show in §7 that profiles learned by FlashProfile may aid PBE based applications, such as Flash Fill [19] for data transformation.

A recent work leverages profiling based on hierarchical clustering, for tagging sensors used in building automation [10]. However, they use a fixed set of features relevant to their domain, and do not qualify clusters with patterns.

Grammar Induction: Syntactic profiling is also related to the problem of learning regular expressions, or more generally a grammar [14] from a given set of examples. Most techniques in this line of work such as L-Star [7] and RPNI [39], assume availability of both positive and negative examples, or a membership oracle. Furthermore, these techniques are either too slow or do not generalize well [9].

Finally, the LearnPADS [16, 51] tool generates a syntactic description and a suite of tools for processing semi-structured data. However it does not support refinement.

Program Synthesis: Our implementation, FlashProfile, uses an inductive program synthesis framework, PROSE [4] (also called FlashMeta [40]). Inductive synthesis, specifically programming-by-examples [19, 21] (PBE) has been the focus of several recent works on automating data-driven tasks, such as string transformations [18, 46, 45], text extraction [29], and spreadsheet data manipulation [20, 22]. However, unlike these applications, data profiling does not solicit any examples from the user. We demonstrate a novel application of a supervised synthesis technique to solve an unsupervised learning problem – our clustering technique drives the synthesizer by creating examples as necessary.

9. CONCLUSION

With increasing volume and variety of data, we require better profiling techniques to enable end users to easily understand, and analyse their data. Existing techniques target simple data-types, mostly numeric data, or only generate partial profiles for string data, such as frequent patterns. In this paper, we present a technique for learning comprehensive syntactic descriptions of string datasets, which also allows for interactive refinement. We implement this technique as FlashProfile, and present extensive evaluation on its accuracy and performance. We show that profiles generated by FlashProfile are useful both for manual data analysis and in existing PBE systems.

10. REFERENCES

[1] Ataccama One Platform. https://www.ataccama.com/.
[2] Microsoft SQL Server Data Tools (SSDT). https://docs.microsoft.com/en-gb/sql/ssdt.

[3] OpenRefine: A free, open source, powerful tool for working with messy data. http://openrefine.org/.

[4] Program Synthesis using Examples SDK. https://microsoft.github.io/prose/.

[5] Trifacta Wrangler. https://www.trifacta.com/.

[6] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *The VLDB Journal*, 24(4):557–581, 2015.

[7] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[8] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[9] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. *arXiv preprint arXiv:1608.01723*, 2016.

[10] A. A. Bhattacharya, D. Hong, D. Culler, J. Ortiz, K. Whitehouse, and E. Wu. Automated metadata construction to support portable building applications. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, pages 3–12. ACM, 2015.

[11] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[12] T. Dasu and T. Johnson. Hunting of the snark: Finding data glitches using data mining methods. In *IQ*, pages 89–98. Citeseer, 1999.

[13] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 240–251. ACM, 2002.

[14] C. De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[15] X. L. Dong and D. Srivastava. Big data integration. *Proc. VLDB Endow.*, 6(11):1188–1189, Aug. 2013.

[16] K. Fisher, D. Walker, and K. Q. Zhu. Learnpads: automatic tool generation from ad hoc data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1299–1302. ACM, 2008.

[17] W. H. Gomaa and A. A. Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13), 2013.

[18] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[19] S. Gulwani. Synthesis from examples. In *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, volume 10, 2012.

[20] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[21] S. Gulwani, J. Hernandez-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[22] S. Gulwani and M. Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 803–814. ACM, 2014.

[23] S. Hanneke. Theory of disagreement-based active learning. *Found. Trends Mach. Learn.*, 7(2-3):131–309, June 2014.

[24] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[25] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[26] L. Kaufman and P. Rousseeuw. *Clustering by means of medoids*. North-Holland, 1987.

[27] D. Kini and S. Gulwani. Flashnormalize: Programming by examples for text normalization. In *IJCAI*, pages 776–783, 2015.

[28] T. Lau. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine*, 30(4):65, 2009.

[29] V. Le and S. Gulwani. FlashExtract: A framework for data extraction by examples. In *ACM SIGPLAN Notices*, volume 49, pages 542–553. ACM, 2014.

[30] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[31] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 21–30. Association for Computational Linguistics, 2008.

[32] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[33] S. Lohr. For big-data scientists,janitor workis key hurdle to insights. *New York Times*, 17, 2014.

[34] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[35] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[36] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys (CSUR)*, 20(3):191–221, 1988.

[37] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 291–301. ACM, 2015.

[38] F. Naumann. Data profiling revisited. *ACM SIGMOD Record*, 42(4):40–49, 2014.

[39] J. Oncina and P. García. Identifying regular languages

in polynomial time. *Advances in Structural and Syntactic Pattern Recognition*, 5(99-108):15–20, 1992.

[40] O. Polozov and S. Gulwani. FlashMeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

[41] O. Polozov and S. Gulwani. Program synthesis in the industrial world: Inductive, incremental, interactive. In *5th Workshop on Synthesis (SYNT)*, 2016.

[42] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM Sigmod Record*, volume 25, pages 294–305. ACM, 1996.

[43] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.

[44] C. Scaffidi, B. Myers, and M. Shaw. Intelligently creating and recommending reusable reformatting rules. In *Proceedings of the 14th international conference on Intelligent user interfaces*, pages 297–306. ACM, 2009.

[45] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827, 2016.

[46] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5(8):740–751, 2012.

[47] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. Gordian: efficient and scalable discovery of composite keys. In *Proceedings of the 32nd international conference on Very large data bases*, pages 691–702. VLDB Endowment, 2006.

[48] T. Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons. *Biol. Skr.*, 5:1–34, 1948.

[49] W. E. Winkler. The state of record linkage and current research problems. In *Statistical Research Division, US Census Bureau*. Citeseer, 1999.

[50] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[51] K. Q. Zhu, K. Fisher, and D. Walker. Learnpads++: Incremental inference of ad hoc data formats. In *International Symposium on Practical Aspects of Declarative Languages*, pages 168–182. Springer, 2012.