Discovering Concepts Using Large Table Corpus

Keqian Li*
Univ. of California, Santa Barbara
klee@cs.ucsb.edu

Yeye He
Microsoft Research
yeyehe@microsoft.com

Kris Ganjam
Microsoft Research
krisgan@microsoft.com

ABSTRACT

Existing work on knowledge discovery mostly uses natural language techniques to extract entities and relationships from textual documents. However, today relational tables are abundant in quantities, often with clean and well-structured data values. So far these rich relational tables have been largely overlooked for the purpose of knowledge discovery. In this work, we study the problem of extracting concept hierarchies given a large table corpus. Our method first iteratively groups values in a table corpus based on co-occurrence statistics to produce a candidate hierarchical tree. The hierarchy of the tree is then summarized by selecting nodes that can best “describe” the original corpus, in order to produce a small tree with desired concept hierarchies, and is easy for humans to understand and curate. We design our algorithms based on map-reduce to scale to large table corpus. Experiment evaluation on real enterprise table corpus shows that proposed approach can generate concepts with high quality.

ACM Reference format:
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Existing research on knowledge discovery [2, 5, 15, 21, 22, 26] mostly relies on textual documents corpus and the use of natural language techniques. This has led to a long and fruitful line of research, and resulted in many influential knowledge bases such as NELL [5], YAGO [21], Probase [26], etc. However, we observe that so far the success has been limited to the public web domain. In addition to this one single web domain that has been heavily studied, there also exist numerous enterprise domains, each with its own entities and concepts that share little overlap between enterprises (e.g., each enterprise will have its own product categorizations, cost-center classifications, and organizational hierarchies, etc.). Just like in the web domain, knowledge discovered from enterprise domains would bring tremendous benefits for applications such as keyword search and document ranking [23], data integration [3], data cleaning [13], etc.

Despite the success of natural language based techniques in the web domain, our experience suggests that they do not work well when applied to enterprise domains for two reasons. First, text documents are scarcer in enterprises than in the general web. While text data on the web typically have thousands of mentions of common sense knowledge in text forms, e.g., “countries such as USA, Canada, ...”, in enterprises mentions like this are less frequent. Using Microsoft’s intranet search engine, we only found a few hits for the text pattern “Microsoft products such as Microsoft Office 2016, ....”. Replacing “Office 2016” with less popular names such as “SQL Server 2016” would yield no hits. We believe that not only are text data in enterprises orders of magnitude less in quantity than the general web, relevant information are also less likely to be represented in plain English texts.

The second drawback of natural language based techniques is that they tend to generate instances that are not entirely compatible with each other. For instance, in addition to extracting “Microsoft Office 2016” from the sentence above, such techniques would also extract “Office 2016 profession plus” (from a sentence “Microsoft products such as Office 2016 profession plus, ...”), and many other mentions like “Office 2016”, “Office”, “Excel 2016”, etc. These entity instances may be at different conceptual levels and not consistent together, due to the idiosyncrasy in how people mention these entities. Entity-concept relationships so discovered would require substantial post-processing and cleaning before they can be used. Given that each enterprise has its own proprietary data, this is very expensive to curate and difficult to scale.

To overcome these issues, we observe that an alternative is to leverage tabular relational data that have become ubiquitous in enterprises (in the form of enterprise spreadsheets and database tables), as well as on the web as web tables. In Microsoft intranet for example, we crawled and extracted over 500K tables from enterprise spreadsheets, each of which has hundreds of rows and tens of columns, and covers a wide variety of topics. Furthermore, data values in these tables are often clean and well-structured. For example, values in the same column often are a coherent set of related entities in the same concept (e.g., list of products, list of cost center names, etc.). Table 1 shows a few such examples. Because spreadsheets are typically well-structured and clean, it is unlikely that incompatible entities like “Office”, “Office 2016”, “Excel 2016” and “Office 2016 profession plus” would be mixed in the same column.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Example Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit Center</td>
<td>P73670 US - DPE Central Mgmt., P19844 US-HealthCare Provider-Payor, ...</td>
</tr>
<tr>
<td>Data Center</td>
<td>39995-Canyon Park, 40422-San Antonio IDC1, 40787-Amsterdam, 40330-Tukwila 5, ...</td>
</tr>
<tr>
<td>Windows Protocol</td>
<td>MS-WSUSOD, MS-NETOD, MS-AUTHSOD, MS-CAPR, MS-ISTD, MS-DRPEI, ...</td>
</tr>
</tbody>
</table>

Table 1: Example concepts and entity instances extracted from an enterprise table corpus.
The abundance of rich relational tables and their structured nature makes them suitable for knowledge discovery in enterprises. However, as existing techniques on knowledge discovery focus almost exclusively on using natural language texts [2, 5, 15, 21, 22, 26], using relational tables for knowledge discovery is a topic that has been largely overlooked in the existing literature.

We in this work take the first step towards the direction of using relational tables for knowledge discovery, and study the problem of discovering coherent sets of entity instances that collectively form concepts. It is worth noting that while HTML tables have been used for set expansion [1, 12, 20, 24], these techniques produce one set of entities at a time, and require concept names and/or seed entities as input. In this work, we attempt to utilize a large table corpus to discover many concepts simultaneously without seed entities or concept names.

The challenge of using tables for concept discovery is that while tables are structured, they still cannot be used directly. For example, entity instances in a column may be incomplete – i.e. it may only contain a subset of instances in that concept. Furthermore, while most table columns have coherent sets of entities, some columns can still have irrelevant values. For instance, some columns may be mixed with multiple related concepts (e.g., both cities and countries may be used for the “address” column), or certain values extracted in a column may just be some meta-data (e.g., column headers, section headers, and tally entries such as “sum” and “total”).

Our main insight is to leverage co-occurrence statistics of values from a large table corpus. While individual table columns may have irrelevant values such as meta-data entries, because the same concept are occurring repeatedly in the corpus with substantial redundancy, these outlier values can be identified as having low statistical correlation with true members of that concept (e.g., cities vs. countries, or cities vs. “total”). Furthermore, even if many columns are incomplete, different subsets of entities from the same concept often have sufficient co-occurrence that allows values in different columns to be merged, producing clusters of values corresponding to different concepts. The resulting concepts can be manually curated like knowledge bases constructed for the Web, and used to benefit a variety of applications such as master data management [9], document ranking [23], data integration [3], etc.

**Contribution** Our work makes the following contributions.

- We studied the novel problem of automatic concept discovery from table corpus, without using concept names or seed instances.
- We proposed a data-driven approach that discover concepts and their hierarchies automatically. Our results are easy for humans to understand and amenable to curation. Our algorithm is designed to run on map reduce to scale to large table corpus.
- Our evaluation using real table corpus extracted from enterprise spreadsheets, and human labeled benchmarks suggests that the proposed approach can discover concepts with high quality.

**2 SOLUTION OVERVIEW**

At a high level, our end to end problem can be stated as follows: given an input table corpus consisting of a set of *table columns* $C$, where each column $c \in C$ is a set of values in the universe of possible values $V$ in the corpus, or $c \subseteq V$. The goal of concept discovery is to group values in $V$ to form coherent clusters. We denote the set of clusters as $O = \{o|o \subseteq V\}$, where each cluster $o$ consists of a set of values and may correspond to a real-world concept. Note that in this work we only produce clusters of values in different concepts, and do not attempt to generate textual “names” to summarize these concepts. Producing concept names based on instance values (e.g., [7]) is a somewhat orthogonal topic that warrants additional studies.

Our approach, called Concept Construction from Coherent Clusters (C$^4$), has two main steps. In the first step, we compute statistical co-occurrence scores between all values in the table corpus. We then iteratively merge related values into clusters in a bottom-up, hierarchical manner. This is because natural concepts follows a tree hierarchy [16], starting from the narrow concept in the leaf nodes, up to root levels with super-categories and broader concepts. We iteratively merge values closely related to each other first, which in later iterations will expand to include less coherent values. This results in a clustering dendrogram, with a large number of coherent clusters in a deep hierarchical tree. We denote it by a node set $O_c = \{o|o \subseteq V\}$, where each node $o$ is characterized by the set of values, i.e. $o \subseteq V$, and an edge set $E_c \subseteq \{(o_1, o_2)|o_1 \subseteq o_2, o_1 \in O_c, o_2 \in O_c\}$ that represents containment relationships. We call the resulting tree $(O_c, E_c)$ the *deep clustering tree* that will be used as input for the second step. The challenge we face here is to scale bottom-up clustering to a large graph, with over 10M entity nodes (each representing a distinct value in table cells). For this we adapt the random-mate algorithm for computing connected components [19]. This step produces necessary clusters for the second stage – we use known techniques here and make no new algorithmic contribution.

In the second step, we take the deep hierarchical tree $(O_c, E_c)$ and select a subset of nodes $O \subseteq O_c$ that likely correspond to good concepts to form the final output. This corresponds to the process of reducing the deep tree into a much shallower tree. We formulate tree-reduction as an optimization problem, which finds the subtree of fixed depth that can best describe the original table corpus. The rationale here is that such a sub-tree will likely have much smaller tree size while still being able to describe much the original table corpus, thus consistent with the general principle of minimum-description-length. We propose a parallel dynamic programming algorithm that solves the problem optimally and scales to large table corpus. The resulted concepts set carries the additional benefits of having a hyponym-hypernym tree structure, which makes it easy for humans to both inspect and curate.

We illustrate our two stage approach with an example in Figure 1.
and US) based on continents, and finally into ATUs in the world. This process produces a large number of intermediate clusters, many of which do not correspond to useful concepts. In the second stage, our clustering tree reduction algorithm will select nodes to form the final concept tree that corresponds to commonly used concepts, which reduces the size of the clustering tree and make it easy for humans to curate (Figure 1b).

By first exhaustively enumerating clusters and then optimally trimming down the clusters size, our algorithm aims to generate concepts that can achieve a high recall rate allowed larger size of output, and retain the best quality clusters when the output size is limited.

3 CANDIDATE CLUSTER GENERATION

In this section we describe the first step of our algorithm, which is to generate an extensive set of initial clusters that become candidates for pruning. Since we mostly adapt known techniques in this step, we discuss it here for completeness and do not consider it as our key contributions.

Given a large table corpus, we use statistical value co-occurrence to compute similarity between any two values to capture their semantic relatedness. For example, in Figure 1, if a value (e., “France.02.SVCS-RLCT”) co-occurs frequently with another (e., “Germany.06.MNF-EU”), in the same columns, then intuitively they will have a high similarity score. We can thus use a corpus-driven approach to define similarity \( s : V \times V \to (0, 1) \) for each pair of values. An instantiation of this using Jaccard can be defined as follows.

**Definition 1 (Value Similarity by Jaccard).** The similarity score between two table value \( v_1, v_2 \) using Jaccard similarity can be defined as

\[
s(v_1, v_2) = \frac{|c[v_1] \cap c[v_2]|}{|c[v_1] \cup c[v_2]|}.
\]

In principle, any set-based or vector-based similarity metric such as Point-wise Mutual Information, Dice coefficient, Jaccard containment would all work here. We found the Jaccard coefficient works the best in this problem.

Once we have similarity between pairs of values, we can start producing initial clusters. We tried various different clustering approaches, including single-link, average-link, complete-link, correlation clustering, density-based clustering, etc. Average-link produces initial clusters of superior quality, likely because it is less sensitive to anomalous co-occurrence scores. We also conclude that techniques like single-link and correlation clustering are non-starters for our data sets, based on the quality of the initial clusters they produce.

The problem with average-link is that it is of complexity \( O(n^3) \), which is difficult to scale to large data sets (in our graph we have \( n > 10M \)). Our method remedies this by efficiently partitioning the data and reducing the clustering rounds achieving both space and time efficiency.

Our map-reduce based clustering can be viewed as a batch version of agglomerative clustering. The idea is that instead of selecting only one pair of nodes to merge in one iteration as in traditional agglomerative clustering, we will merge multiple values in each round. In particular, we collapse all node pairs whose edge scores are higher than a selected similarity threshold. This can be achieved by thresholding the edge weights and computing connected component. We can gradually lower the threshold in subsequent iterations. In an extreme case, this is equivalent to vanilla agglomerative clustering if we lower the similarity threshold just enough to only merge one pair of nodes in every iteration.

We formally state the algorithm in Algorithm 1. At each clustering iteration \( i \), the algorithm will maintain a graph of supernodes \( O^i \), each is a cluster consisting of one or more nodes in the original input \( V \). It will use a a threshold \( \theta_i \) to determine the batch size, where the edge between any pair of nodes with similarity above \( \theta_i \) will be merged in the next round (line 5). The threshold itself is computed as the top \( k \) score among all edge similarities (line 7).

Algorithm 1 Exhaustive hierarchical concept enumeration (\( i : V \times V \to R, V, k \))

1: \( O^0 \leftarrow V \)
2: \( E^0 \leftarrow \{ (v_1, v_2), s(v_1, v_2)) \mid (v_1, v_2) \in O^0 \times O^0 \} \)
3: \( \theta^0 \leftarrow \text{SetThreshold}(E^0, 1), i \leftarrow 0 \)
4: while \( \theta^i > 0 \) do
5: \( E_{Merge} \leftarrow \{ (v_1, v_2) | v_1, v_2, s(v_1, v_2) \in E^i, s(v_1, v_2) > \theta^i \} \)
6: \( O^{i+1}, E^{i+1} \leftarrow \text{ConnectedComponents}(O^i, E_{Merge}) \)
7: \( \theta^{i+1} \leftarrow \text{TopK}(E^{i+1}, k) \)
8: \( i \leftarrow i + 1 \)
9: end while
10: return \( \bigcup O^i \)

We can show the following complexity result for this procedure.

**Theorem 1.** For a chosen a batch size of \( k \), the number of distributed rounds will be \( O\left(\frac{W \log k}{k}\right)\).
A proof for this can also be found in [14].

4 CONCEPT SELECTION

In the next stage, given the deep clustering tree \((O_c, E_c)\), our aim is to select the best quality clusters \(O \subseteq O_c\) as the final output. We will follow a data driven approach, propose the use of clustering of values to describe original table column and use that as the quality measure, formulate the problem as maximizing the quality measure subjected to size constraint, and provide theoretical analysis (Section 4.1), and then provide an optimal solution that is parallel by design and runs on the map reduce framework (Section 4.2).

4.1 Problem Formulation

In this section we will formally define the objective as well as constraints for our task of concept selection. We measure the quality of the generated clusters based on the following key observation: although data noise commonly exists in the table corpus, there will still be less noisy columns that contain a nearly complete and clean set of values that maps to a concept, thus a cluster that has table columns closely matching its contents will be evidence for that cluster to be a high quality cluster that maps to real concepts. We formalize the above notion as “corpus coverage” and formalize it as follows [11]:

**DEFINITION 2 (CORPUS COVERAGE).** We say that a node \(O \in O_c\) in the clustering tree can “describe” a table column \(c \in C\), if the Jaccard similarity measure between the concept node and the table columns is above a threshold that’s predefined. Formally, we denote a function \(d : O_c \rightarrow C\) as the set of table columns a node can describe, which we call a “corpus coverage” for a node in the clustering tree. Furthermore, for a set of nodes \(O \subseteq O_c\), the “corpus coverage” of \(D(O)\) will be the union of the corresponding node level corpus coverage: \(D(O) = \{\bigcup d(o) | o \in O\}\).

In order to effectively select the nodes in the clustering tree that help with the overall corpus coverage and remove ones that are redundant, we impose a size constraint to ensure the resulted selection have a simpler structure compared to the original deep hierarchical tree. We will start by providing a key insight for the processing of selecting nodes from a tree: If a set of nodes are selected from a tree, they will themselves form a tree structure, which we call the “concatenated subtree”:

**DEFINITION 3 (CONCATENATED SUBTREE).** Given a tree with nodes \(V\), edges \(E\), a subset of nodes \(V' \subseteq V\), a concatenated subtree induced by the subset of nodes, \((V_S, E_S)\), can be obtained as follows. First, we include all \(V'\) into \(V_S\); obtain the “hierarchy” by including all such edges connecting nodes \(u \in V'\) to its closest ancestor that is also in \(V'\) into \(E_S\), and finally to make it a proper tree, we add a dummy node \(v_0\) as root of the tree and connect all nodes that don’t have a parent to it.

The above definition points out a key fact: selecting a subset of nodes from a tree is equivalent to selecting a subtree. Our task thus becomes selecting a subtree of clusters from the original clustering tree. We identify “depth” as the key measure for the simplicity of the resulted subtree: no matter how many nodes a tree of clusters may have in a particular layer, they will always form one single partition over the set the original elements to be clustered, but a tree of clusters with higher depth means there are many repeated layers of partitioning. This redundancy can also be seen from the end user’s view: if he is looking up the concept using one specific instances, the resulted concept tree would return many clusters containing it, one of them may be the desired concepts, while the rest are noisy clusters that either lack the necessary elements or contain extra elements. A clustering tree of higher depth would contain more of such noise.

We therefore take “target coverage” as objective, “depth” as constraint, and formulate our concept selection problem as a more general problem of subtree selection for maximum target coverage.

**PROBLEM 1.** (Maximum Target Coverage Subtree selection (MTCS)). Given an input tree \((V, E)\), a set of targets \(T\), a target coverage function \(d : V \rightarrow 2^T\), a height constraint \(H\), our aim is to select a set of tree nodes \(V_S \subseteq V\), so that the concatenated subtree \(V_S, E_S\) induced by \(V_S\) has height as most \(H\) and the target coverage \(\bigcup_{v \in V_S} d(v)\) is maximized.

For our task of selecting concepts from clustering tree, the tree in the MTCS problem corresponds to the deep clustering tree \(O_c, E_c\) generated in the previous stage, targets correspond to the set of table columns \(C\), coverage function will be the predefined corpus description function \(d : O_c \rightarrow C\), and the output corresponds to the selected concepts \(O\).

![Figure 2: Continuation for the example in Figure 1. Formatted for illustrating the clustering tree reduction stage.](image)

We illustrate the above notion with a running example as shown in Figure 2.

**EXAMPLE 2 (MTCS PROBLEM).** From the annotated candidate concept tree in Figure 2, we can observe the desired output concept hierarchy: the concept of all ATUs in the world \(a\), and the six concept of ATUs for six continents: \(b_1, b_2, b_3, e, i, j\). By formulating it as the MTCS problem, we’re able to capture the joint set of nodes \(a, b_1, b_2, b_3, e, i, j\) (with corpus coverage of 15) as the optimal solution, under the height constraint of 2.

The above problem is APX-hard, which can be established through a gap-augmenting reduction from Set Cover problem.
4.2 Dynamic Programming Algorithm

In this section we present our dynamic programming style algorithm that optimally solves the MTCS problem while being scalable to large corpus.

Our solution deviates from typical dynamic programming algorithm because the problem is shown to be APX-hard and designing an exact version of dynamic programming is not possible. We address this challenge by providing a more exhaustive scheme of dynamic programming, described as follows. Typically, a dynamic programming algorithm will have the following structure: one will define a hierarchy of subproblems, obtain an optimal solution for the initial cases and then iteratively find optimal solution for some subproblems from the optimal solutions to all their children cases when they’re available. Our scheme is different in that we maintain not one single optimal solution to each subproblem, but a set of possibly optimal solutions for each subproblem, and each time derive the set of possibly optimal solutions for a parent case based on all unique combinations of set of possibly optimal solutions from each of the children cases. Alternatively, we can treat the task of finding all possible solutions that could lead to global optimal solution as one simple problem, and continue with normal dynamic programming induction. We adopt this view and state the subproblem as below

Problem 2. (Subproblem to Maximum Target Coverage Subtree selection (S-MTCS)). The subproblem, given all the original input to the MTCS problem together with a particular tree node \( v_S \in V \) and particular height constraint \( h \), we call the descendants tree as the subtree formed by \( o \) and all its descendants. Our task will be to find not one single SubTree selection (ST), but the whole set of Possibly-optimal SubTree selection (PST) with respect to the descendants tree. More specifically, the set of Possibly-optimal SubTree selection is defined as all the ST selection that is possible to form a part of global optimal solution to MTCS problem, with the possibility taken over all the possible node selection choices made outside of the descendants tree.

Representation of a single SubTree selection (ST). The representation of a single SubTree selection is crucial for distinguishing a tree reduction that is possible to form a part of global optimal solution from the one that is not. It is motivated by viewing the solution from each single target’s perspective: when it will get covered by the selected nodes, when it is impossible to be covered.

We exploit this property of the our target:

Definition 4 (Tree coverage). Given all the original input for the MTCS problem, the tree coverage for a target \( t \in T \) is defined as \( d^{-1}(t) = \{ v | v \in V, t \in d(v) \} \).

We describe the following three categories of targets based on their tree coverage, that reflects the difficulty of processing them when we built the induction rule.

Definition 5. (Lineage Target, Partial Lineage Target, Non Lineage Target). Given all the original input for the MTCS problem, we define a lineage as the node set of a directed path that connects a tree node to its direct descendent. A target \( t \in T \) is a lineage target if its tree coverage \( d^{-1}(t) \) is a lineage. A target \( t \in T \) is a partial lineage target if its tree coverage \( d^{-1}(t) \) is a subset of lineage, excluding lineage. A target \( t \in T \) as non lineage target it is neither a lineage target nor a partial lineage target.

We can thus parameterize each targets differently: for a lineage target, we can fully characterize them using \( \text{start} \), the lowest grandchild in the path, and \( \text{end} \), the highest grandparent in the path. For a partial lineage target, say that lineage with shortest path of which the target coverage is a subset has \( \text{start} \) as the lowest grandchild in the path, \( \text{end} \) as the highest grandparent in the path, and \( N \) being the set of nodes themselves, we denote \( \text{missing} = N - d^{-1}(t) \), and characterize the target using (\( \text{start} \), \( \text{end} \), \( \text{missing} \)). For a non lineage target, we assign an unique id and do not attempt to parameterize them.

Intuitively, the lineage target corresponds to table columns for normal concepts that is matched to a series of gradually growing clusters of table values in the deep clustering tree, and partial lineage target corresponds to the case when the mapping is "flawed" that there is some abnormal clusters that become dissimilar in the gradually growing set of clusters, finally the non lineage target corresponds to the abnormal case where highly abnormal table columns that maps to two disjoint clusters of table values.

Next we proceed to characterize a single ST using the above result. As the most targets that is already covered will not play a role in the future induction process, we will store them as a number using a counter. What we need to keep track of are the ones that is possible to be covered in the future induction: first they hasn’t been covered by current ST, and they need to show possibility of being covered in the future, via a simple test: the height of the current node of concern \( v_S \), need to be between the lowest height and highest height of the tree nodes in the target’s tree coverage. We will call these target active. Finally, we treat non-lineage target as an exception and always use the set of covered targets to keep track of them. We organize this the following definition:

Definition 6 (Subtree Selection). A Subtree Selection (ST) will be represented as: 1) a count for all lineage and partial lineage target that’s already covered by the selected nodes covered 2) a dictionary \( \text{dict}_{1} \), keyed by (\( \text{start} \), \( \text{end} \)), storing the count of active lineage targets of corresponding entry. 3) a dictionary \( \text{dict}_{PL} \), keyed by (\( \text{start} \), \( \text{end} \), \( \text{missing} \)), storing the count of active lineage targets of corresponding entry. 4) finally a set \( \text{covered} \) storing the unique id for non lineage targets that’s covered by the tree selection.

Representation of Possibly-optimal SubTree selection (PST). As we mentioned earlier, we need to store the set of all possible subtree selection that is possible to form global optimal solution. To implement this policy, we keep all ST for a subproblem as PST, if they pass a simple test: if it is not obviously dominated by another ST according to a predefined ordering \( <_{ST} \) that explicitly forbid it to be part of global optimal solution, we will throw it out of PST. The \( <_{ST} \) is implemented as follows. We say \( ST_1 <_{ST} ST_2 \), if if for \( ST_1 \) all the “active” targets (values in \( \text{dict}_{1} \), \( \text{dict}_{PL} \), \( \text{dict}_{N} \)) is converted into the covered counts \( covered \) and that’s still below \( ST_2 \), we can say \( ST_1 <_{ST} ST_2 \). We will define \( PST_{\text{h},k} \) as the set storing all such ST for the subproblem \( S - \text{MTCS} \) node \( v \) in the tree and a height constraint \( h \).

Induction rules. Since the solution to subproblem is to find PST with respect to a node \( v \) and height constraint \( h \), the induction rule will be with respect to each \( PST_{v,k} \). As in the dynamic programming scheme we will derive all relevant PST from each of its children nodes \( \{ j | j \text{ is a child of } v \} \). On a higher level, we will derive each
candidate ST from each unique combinations of the STs in the PSTs from each of its children, and then select one that is not dominated according to the partial ordering requirement \(<_{ST}\) as stated above.

The derivation for each individual ST in \(PST_{v, h}\) will either one of the two scenarios, similar to many dynamic programming scheme: the first case where we don’t include the current node into selection, deriving an ST from the \(\{PST_{j_1, h}\} | j_1 \text{ is a child of } v\) and the second case where we do include the current node, deriving an ST from the \(\{PST_{j_1, h-1}\} | j_1 \text{ is a child of } v\). The operation for both cases can be summarized as four steps: 1) an derive\_from\_children operation that sums up the corresponding entries for \(dict_{L}, \text{dict}_{PL}\), sums up covered, merges \(covered_{NL}\) for the ST coming from each of the children nodes (line 16 in Algorithm 2), 2) an add\_newly\_active\_targets operation to add newly active targets \(dict_{L}, \text{dict}_{PL}\) which will be those with the current node \(v\) as their start nodes, expect for non lineage which become active whenever the current node \(v\) is within the height of the lowest and highest nodes of its tree coverage (line 17). 3) an cover\_targets operation if we will include current node \(v\) into the selected nodes ST, where we sum all the counts from matching entries in \(dict_{L}, \text{dict}_{PL}\) into the count covered, remove those entries, and merge the non-lineage targets of coverage by the current nodes into \(covered_{NL}\) (line 18-20). 4) an discard\_inactive\_targets operation to discard entries in \(dict_{L}, \text{dict}_{PL}\), who won’t be "active" in the future, i.e. current node \(v\) as the same height as highest nodes in their tree coverage (line 21).

On an abstract level, the induction can be briefly described with the formula below. A more detailed description of the process is shown in Algorithm 2.

\[
\begin{align*}
\{l_1,l_2, \ldots, l_k\} &= \text{[children nodes of } v] \quad (1) \\
S_1 &= \{\text{notIncludeCurrent}(l_1, l_2, \ldots, l_k) | \{(l_1, l_2, \ldots, l_k) \in \text{PST}_{j_1, h} \times \text{PST}_{j_2, h} \times \text{PST}_{j_k, h} \} \} \quad (2) \\
S_2 &= \{\text{includeCurrent}(l_1, l_2, \ldots, l_k) | \{(l_1, l_2, \ldots, l_k) \in \text{PST}_{j_1, h-1} \times \text{PST}_{j_2, h-1} \times \text{PST}_{j_k, h-1} \} \} \quad (3) \\
\text{notIncludeCurrent} &= \text{trim}(S_1 \cup S_2) \quad (4)
\end{align*}
\]

\(\text{notIncludeCurrent}\) corresponds to the induction for scenario of not including the current node in the ST, as shown in case 1 above, (line 15-20 in Algorithm 2). \(\text{includeCurrent}\) corresponds to the scenario of including the current node in the ST, as shown in case 2 above, (line 22-28 in Algorithm 2) \(\text{trim}\) corresponds to the operation of discarding the ST that’s inferior to others in partial ordering comparison \(<_{ST}\) (line 29).

We illustrate the approach using an example.

**Example 3 (MTCS algorithm).** The optimal solution for the MTCS with height constraint 2 in Figure 2, represented as \{(covered = 15, dict_{L}, dict_{PL}, covered_{NL} = empty\}, will be captured by our approach as shown below: it will be solved by combining an ST from \(\text{PST}_{b_1, 0}\), and one from \(\text{PST}_{c_1}\). For the one from \(\text{PST}_{b_1, 0}\), it comes from an ST in \(\text{PST}_{b_1, 0}\), which traces back to an ST in \(\text{PST}_{b_0, 0}\). For the ST from \(\text{PST}_{c_1}\), it comes from combining an ST in \(\text{PST}_{c_1, 1}\), which traces back to an ST in \(\text{PST}_{c_0, 0}\); and one ST in \(\text{PST}_{c_1, 0}\) which traces back to an ST in \(\text{PST}_{c_1, 1}\), and one in \(\text{PST}_{c_1, 1}\). It can be verified they are not inferior by the ordering \(<_{ST}\) so they got propagated up. From these traces we can recover the optimal nodes selection, as marked in the figure.

**Algorithm 2** Algorithm for MTCS

1. **INPUT:** tree \((V, E)\), target set \(T\), target coverage function \(d : V \rightarrow 2^T\), height constraint \(H\)
2. **OUTPUT:** selected nodes \(V_S \subseteq V\)
3. **for** node \(v\) with node height 0 **do**
   4. \(T \leftarrow \text{add\_newly\_active\_targets}(v)\)
   5. **if** the combination comes from height constraint \(h - 1\) **then**
       6. \(T \leftarrow \text{cover\_active\_target}(T, v)\)
   7. **end if**
   8. **ST \leftarrow discard\_inactive\_targets}(T, v)\)
   9. **end for**
10. **for** stage from 0 up to the height of the original tree **do**
    11. **for** node \(v\) with node height stage **do**
    12. \(S = \emptyset\)
    13. \(\{u_1, u_2, \ldots, u_k\} \leftarrow \text{children of } v\)
    14. **for** height constraint \(h = 0 \ldots H\) **do**
    15. **for** combination \((ST_{1}, ST_{2}, \ldots, ST_{k}) \in \text{PST}_{u_1, h} \times \text{PST}_{u_2, h} \times \text{PST}_{u_k, h}\) **do**
        16. \(T = \text{derive\_from\_children}(ST_{1}, ST_{2}, \ldots, ST_{k})\)
        17. \(T = \text{add\_newly\_active\_targets}(T, v)\)
        18. **if** the combination comes from height constraint \(h - 1\) **then**
            19. \(ST = \text{cover\_targets}(ST, v)\)
        20. **end if**
        21. \(ST \leftarrow \text{discard\_inactive\_targets}(T, v)\)
        22. \(S \leftarrow S \cup ST\)
    23. **end for**
    24. \(ST_{V, h} \leftarrow \{l | l \in S, \forall i \neq l, l <_{ST} l\}\)
    25. **end for**
    26. **end for**
    27. **end for**
    28. \(ST_{OPT} \leftarrow \{ST | ST \in \text{PST}_{\text{root of tree}(V, E), H} | ST\text{ has maximum number of covered targets}\}\)
    29. \(V_S \leftarrow \text{backtrack}(ST, V, E, \{PST(v, h)\})\)
    30. **return** \(V_S\)

**Algorithm Description.** We formally describe our algorithm for solving the MTCS problem in Algorithm 2. We first perform initialization to obtain \(PST\) for leaf nodes, we then invoke procedure \(\text{add\_newly\_active\_targets}\) (line 4), \(\text{cover\_targets}\) (line 5-7), and \(\text{discard\_inactive\_targets}\) (line 8) similar to the induction rule to obtain each possible \(ST\) to merge into \(PST\). In the main loop (line 10-27), we perform the dynamic programming style induction according to the induction rule, from low height budget to higher height budget, from nodes close to the tree leaf to tree root. The final best \(ST\) is selected as the one with highest count of covered targets \(covered\), and the node selection \(V_S\) will be obtained from dynamic programming backtracking (line 28-30).

The algorithm is easily parallelizable since the computation in each for-loop is independent of each other can distributed and put into a map function, with the outside operation as the reduce operation. In that scenario, the space requirement will be \(O(P)\), with the bottleneck being at the reduce stage. \(P\) being the number of possible solution ST in a PST, assuming we can store each ST in constant space. The time complexity, measured in terms of distributed rounds, will therefore be the \(O(P \cdot N)\), \(N\) being the height of input tree. A
5 EXPERIMENT
In this section we present our experimental evaluation of the concept discovery. Our goal is to (1) evaluate the overall accuracy of different clustering approaches; (2) understand the effectiveness of the tree reduction approach; and (3) measure the sensitivity of parameters.

5.1 Experimental Setup
We conduct our experiments on Microsoft Research’s private MapReduce platform. The longest run for the stage of candidate cluster generation took less than 24 hours, while the longest run for the stage of concept selection took less than 3.5 hours.

5.1.1 Dataset. We perform the concept discovery over the dataset of a large enterprise spreadsheet table corpus crawled from Microsoft intranet. There are over 500K tables, 3.2M table columns, 13M distinct cell values, and over 2B edges in the resulting graph. Most entities and concepts are enterprise-specific, as shown by examples in Table 1.

5.1.2 Methods Compared. Connected Components. We implement a straightforward clustering approach for grouping similar table values together based on their co-occurrence graph. In this method, all the highly similar table value pair will be merged into the same concepts. We use a parallel implementation by Rastogi et al. [19].

Correlation Clustering. We implement a parallel correlation clustering algorithm [6] on the map-reduce platform. The algorithm works by sampling among all nodes a number of pivots and grow clusters around them in each round of clustering.

Density Clustering. Density clustering methods such as DBSCAN is another clustering variant that is robust against noisy nodes. We implement the DBSCAN density clustering using a map-reduce programming paradigm: at each round of clustering, we perform a reduce on each node to compute the density and neighborhood size to determine whether it is “core” or “edge”. Then we iteratively find the reachable region for each core and perform the merge. We set the minimize size of a cluster as 5.

Complete Linkage. We also implemented the complete linkage clustering, where a pair of clusters merge only when all constituent nodes are similar to each other. To implement it on a map-reduce platform, we at each iteration only merge pairs of clusters that satisfy the complete linkage similarity requirement. As the clustering iterations proceed, the resulting clusters will converge to a stable state where each complete clusters stop merging with new clusters. We will take this as the result for complete linkage clustering.

WEBSETS. We compare with the WEBSETS approach proposed in [8]. The algorithm emits for each table a set of consecutive triplets of entities and then merge the triplets together if they have significant overlap in terms of entities they contain and columns they appear in. We use the same parameters values reported in the original paper.

C4. We evaluate our proposed approach of C4. Concept Construction from Coherent Clusters, with the following setting. In the first stage of candidate cluster generation, we run the hierarchical clustering for 20 distributed rounds, resulting in a candidate concept tree with height 20. In the second stage of concept selection, we set the height constraint for clustering tree reduction as 1 and 3. By producing a single layer of clusters, we can compare its performance against other flat single layer clustering methods in a fair setting. By extending the selected clustering tree to 3 layers, we can observe how much performance it can gain by allowing some shallow layers of tree structure. To contrast with the original deep clustering tree, we name the selected subtree of 1 layer as C4 with Height 1 Pruning, the selected subtree of 3 layers as C4 with Height 3 Pruning, and the original deep clustering tree as C4 without Pruning. The key parameter in our approach is the threshold value that will determine whether a tree node can describe a column (see definition 2).

5.1.3 Evaluation. Ground Truth Concepts. We constructed a ground truth concepts evaluation set consisting of 100 ground truth concepts with the following procedure: We first randomly sample from the corpus and identify a set of table columns that is close to real concepts, and select 3-5 core entities for that concept. Then using these core entities as seeds we retrieve all the columns in the corpus that contains all of them, merge these tables together, and discard outliers. This will ensure our constructed ground truth concepts are both complete and precise.

Metrics. We adopt the F-score as our measure of accuracy. For each ground truth concept and each method, we will be able to find a cluster generated by that method with highest F1-score, and we will use that as the F1-score for that ground truth concept for that method. The overall performance is taken as the average F1 score of all the ground truth concepts.

5.2 Accuracy Evaluation
5.2.1 Overall Performance. The aggregated performance comparison can be found on Figure 3, showing the overall performance of different methods (see figure 4). We observe that the weakest form of our approach, C4 with height one Pruning, still outperforms all other clustering methods by a large margin. If we allow C4 to have 3-layer hierarchical structure, it will reach a much higher accuracy that is comparable to the original 20-layer deep clustering tree, showing that by removing a majority of the layers of the clustering tree, namely 17 layers, we are still able to retain most meaningful clusters in the clustering tree as measured by the ground truth concepts evaluation.

The effect of different parameter settings for each methods are shown in Figure 4. We can see that a Jaccard similarity threshold of 0.7 is appropriate for determining a mapping relation between a table column and a generated cluster. A threshold that is too high, e.g. 0.9, would be too strict for many generated clusters to map to table columns, while lower threshold, e.g. 0.3, would generate many low-quality clusters table column mapping, which negatively affect the result of concept selection. As for the similarity between table values, we observe a similarity value of 0.5 is suitable for determining if two table values are similar enough to belong to same concept.

5.2.2 Performance by individual concepts. We further inspect the performance for each individual concept as shown in Figure 5. We can see our approaches consistently reach high accuracy over
most of the concepts, while others approaches perform unstably and fail to capture many of the concepts. Furthermore, by comparing our approach that selected subtree height 1,3 and the original tree we can observe that our approach of concept selection indeed retain the true concepts nodes for the majority of the concepts.

5.3 Observation on Corpus Description

In this section, we propose another evaluation measure to further reveal the effectiveness of our concept discovery approach, according to the minimum description length principle. That is, for each approach, we measure the total number of table columns described versus the total number of clusters generated as concepts for each clustering approach, under the Jaccard Similarity threshold 0.7 as the criteria for determining whether a table column is described by the generated cluster. Methods that use fewer generated concepts to describe higher number of table columns are preferred. Table 2 shows the performance for each approach. We can see our method of $C^4$ with height 1,3 Pruning have quite high number of table columns described compared to baseline approaches while using less than 1% of the clusters than the $C^4$ approach without Pruning. They also have the highest ratio between table columns described and concept generated, with the exception of WEBSETS, which is able to identify some very effective clusters as concepts that each can describe many table columns on average, but the number of clusters are too few.

6 RELATED WORK

Digital knowledge bases such as such as YAGO [21], Freebase [4], DBPedia [3] are widely available nowadays and have marked an important advancement of machine intelligence. They aim to build concept hierarchies by integrating from external sources such as Wikipedia as well as crowdsourcing effort, and are mainly concerned with public domain, instead of proprietary enterprise domain. In terms of automatic technique for extracting concepts, most existing concept extraction technique focuses on natural language text [2, 15, 22, 26], where certain language pattern such "X such as Y" are identified and a large collection of subsumption, i.e. is-a relation, can be extracted. These subsumption relationships are then merged together [17, 26] to construct a hierarchical concept taxonomy [16] as outputted in our approach. Their concept discovery methods mainly focuses on detecting "is-a" relationship pattern from natural language texts and their merging the subsumption rely on high accuracy of the collected relationship. In contrast, in table corpus the "is-a" relationship pattern is already given as the value occurrence and the main challenge is to merge values into consistent cluster given the noisy "is-a" relationship between values and table columns.

The high data quality of relational table, and the close resemblance between relational table column and concepts has been underlined by different lines of work exploring the interplay between concept and table columns, and between entities and table values. An important line of work for table corpus is the entity discovery, where they exploit the fact that each table row corresponds to a real world entity and leverage machine learning technique and external knowledge base and search engine to discover and annotate the table rows[18]. On the other hand, the correspondence between table column and concepts has been exploited by the approaches of concept expansion using seed entities: given a set of query instances such as "Tesla" and "Toyota", more car brands will be returned. These methods rely on the cooccurrence of instances in the same list or table columns to build similarity relationship, and based on that perform similarity aggregation methods such as random walk on bipartite graphs to propagate the similarity and return the results [12, 25]. Some approaches use seed instances from multiple concepts to propagate simultaneously, or alternatively use negative instances for concepts [5, 10] to obtain better accuracy. Since it relies on high quality seeds from human, these approaches are limited to the discovery of very small number of concepts. More importantly, it can only operate on concepts that’s already known, not on discovering new concepts from the unknown corpus, thus not suitable for the task of concept discovery.

7 CONCLUSION AND FUTURE WORK

In this work, we study the problem of concept extraction from relational table data, in an large scale setting, without any additional seeds. We propose the method of Concept Construction from Coherent Clusters, that generate candidate concepts to capture possible
Figure 5: Performance by individual concepts

concepts, and then perform concept selection from the generated candidates. Our method is fully implemented on Map-Reduce platform and can process terabytes of corpus in hours. We extensively evaluate several state-of-art large scale concept discovery approaches against our method over the benchmark of the enterprise domain dataset and demonstrate our methods significantly outperform these strong baselines. Many future research directions exist. One would be to integrate natural language data or external knowledge base to provide an holistic framework of concept discovery. Another direction would be to combine more meta-information from the database schema and achieve better curation for the relational tables.

REFERENCES