Scalable Column Concept Determination for Web Tables Using Large Knowledge Bases

Dong Deng[y] Yu Jiang[y] Guoliang Li[y] Jian Li[z] Cong Yu[#]
[y]Department of Computer Science, Tsinghua University, Beijing, China.
[z]Institute for Interdisciplinary Information Sciences,Tsinghua University, Beijing, China.
[#] Google Research, New York, USA
dd11@mails.thu.edu.cn, sunlight07@126.com, liguoliang,lijian83 g@tsinghua.edu.cn, congyu@google.com

ABSTRACT

Tabular data on the Web has become a rich source of structured data that is useful for ordinary users to explore. Due to its potential, tables on the Web have recently attracted a number of studies [6, 18] with the goals of understanding the semantics of those Web tables and providing e ective search and exploration mechanisms over them. An important part of table understanding and search is column concept determination, i.e., identifying the most appropriate concepts associated with the columns of the tables. The problem becomes especially challenging with the availability of increasingly rich knowledge bases that contain hundreds of millions of entities [10, 31].

In this paper, we focus on an important instantiation of the column concept determination problem, namely, the concepts of a column are determined by fuzzy matching its cell values to the entities within a large knowledge base. We provide an e cient and scalable MapReduce-based solution that is scalable to both the number of tables and the size of the knowledge base and propose two novel techniques: knowledge concept aggregation and knowledge entity partition. We prove that both the problem of nding the optimal aggregation strategy and that of nding the optimal

Table 1: Example table from the Web (http://celebrity.psyphil.com/actor/Tom-Hanks/filmography).

1994	Forrest Gump	Robert Zemeckis	Tom Hanks, Robin Wright, Sally Field, ...
1998	Saving Private Ryan	Steven Spielberg	Tom Hanks, Tom Sizemore, Matt Damon, ...
1999	The Green Mile	Frank Darabont	Tom Hanks, David Morse, Bonnie Hunt, ...
1995	Toy Story	John Lasseter	Tom Hanks, Tim Allen, Wallace Shawn, ...
2002	Catch Me If You Can	Steven Spielberg	Tom Hanks, Christopher Walken, Leonardo DiCaprio, ...
2006	The Da Vinci Code	Ron Howard	Tom Hanks, Ian McKellen, Jean Reno, ...

Figure 1: Example knowledge base based on Wikipedia.

to denote the maximum matching, called fuzzy overlap of $\mathtt{T}$ and $\mathtt{C}$, and $|\mathtt{V_C} \widetilde{\cap} \mathtt{E_T}|$ to denote its value.

We can extend exact-matching functions by replacing $\mathtt{V_C} \cap \mathtt{E_T}$ with $\mathtt{V_C} \widetilde{\cap} \mathtt{E_T}$ to define fuzzy-matching functions. Next we take fuzzy Jaccard similarity as an example.

$$\text{Fuzzy Jaccard Similarity: } \textsc{Sim}(\mathtt{C}, \mathtt{T}) = \frac{|\mathtt{V_C} \widetilde{\cap} \mathtt{E_T}|}{|\mathtt{V_C}| + |\mathtt{E_T}| - |\mathtt{V_C} \widetilde{\cap} \mathtt{E_T}|}.$$

For example, consider a column with two cell values {"Shark Night 3D", "Da Vinci Code"} and a type with two entities {"Shark Night", "The Da Vinci Code"}. Suppose we use the Jaccard similarity. "Shark Night 3D" and "Shark Night" are fuzzy matching with similarity 0.67. "Da Vinci Code" and "The Da Vinci Code" are fuzzy matching with similarity 0.75. Thus the fuzzy overlap size is 0.67+0.75=1.42 and the fuzzy Jaccard similarity is $\frac{1.42}{2+2-1.42} = 0.55$.

Scalability Requirement: Since there are hundreds of millions of columns and millions of entities, existing machine learning based methods [18, 27] are neither efficient nor scalable for such large datasets and it calls for scalable methods. To this end, we extend existing MapReduce-based similarity-join methods and develop effective techniques to support efficient and scalable column concept determination, while maintaining the annotation quality.

3. RELATED WORKS

MapReduce: We adopt the MapReduce framework to address the scalability challenge. MapReduce [8] is a dominant framework to support data-intensive parallel computation in shared-nothing clusters. In MapReduce, data is represented as ⟨key, value⟩ pairs and stored in a distributed file system (DFS) across different cluster nodes. MapReduce contains two core functions: `Map` and `Reduce`. The map phase loads and processes different partitions of the input data in parallel and outputs ⟨key, value⟩ pairs. The pairs are then hash-partitioned, sorted by the key, and sent across the cluster in a shuffle phase. The pairs that share the same key are merged in a sorted order and sent to the same reduce node, which then processes the pairs and writes new ⟨key, value⟩ pairs to files on the DFS.

Knowledge Bases: DBPedia [3] and Freebase [5] are similar projects that extract information from Wikipedia to create an extensive structured knowledge base and supplement those data with knowledge that are collaboratively created by thousands of data-loving users. Freebase contained about 2 thousand types and 40 million entities. Probase [31] is a universal, probabilistic taxonomy which is harnessed from billions of Web pages using "isA" and "such as" patterns. Probase includes more than 2.7 million types and 22 million entities. Finally, Yago [25] is a large semantic knowledge base which is derived from Wikipedia and WordNet, and it contains 0.3 million types and 9 million entities.

Annotating Web Tables: Google's WebTables project [6] is the first to start exploring table data on the Web, which has been followed up by a number of other studies [9, 18, 27, 34]. Google has since launched a Table Search project [35] at `http://research.google.com/tables`, where external users can explore the table data. More recently, Google has also started exploring stitching similar tables on the same page [19] to provide better utility from the table data.

At the core of managing Web table data is the annotation problem, i.e., finding meanings for the cells and columns in the table. Existing studies used machine learning techniques to annotate tables [18] that are neither as scalable nor as efficient as the approaches we present in this paper. Other relevant studies (although those do not address the column concept identification problem) include: Yakout et. al. [34] studied how to augment entities with attribute values and discovering attributes using Web tables. Pimplikar et. al. [21] studied how to answer table queries based on column keywords. Quercini et. al. [22] utilized search engines to discover and annotate entities in Web tables. Hignette [15] annotated tables with ontology concepts based on relevance degrees. Wang et. al. [30] utilized the interconnections of Probase to understand Web tables.

Extracting Table Data into RDF Data: There are many studies on extracting table data into RDF data [12, 14]. Guo et. al. [12] used a schema mapping technique to extract and integrate data from tables into RDF. Han et. al. [14] studied how to translate spreadsheet data into RDF. Assem et. al. [26] studied how to annotate and covert quantitative data tables into RDF. Different from these methods, our work focuses on making fuzzy matching based annotation

techniques more scalable and efficient. How to apply our technique to the annotation techniques proposed by those studies is a challenge that we intend to pursue in the future.

Similarity Join: There are many recent studies on similarity join, which, given two sets of objects, find all similar object pairs based on a given similarity function and a threshold [32, 23, 33, 11, 2, 28]. Vernica et al. [28] and Metwally et al. [20] proposed a MapReduce framework to support similarity joins. However they required a similarity threshold and their techniques cannot be extended to support our ranking problem. Kim et al. [17] studied parallel algorithm for top-k similarity join with Euclidean distance constraint using MapReduce while we focus on finding top-k types for each column. These methods do not allow fuzzy matching between entities and cell values.

4. SIMILARITY-JOIN FRAMEWORK

In this section, we extend MapReduce-based similarity joins to support scalable column concept determination. We first take the overlap similarity as an example in Section 4.1 and then extend it to support other functions in Section 4.2.

4.1 Framework for Overlap Similarity

If we use a type to annotate a column, the type should share at least one common entity/cell value with the column. Thus for each column, we first find the types that share common entities with the column and then rank these types to return top-k types. To implement this idea in MapReduce, we introduce a two-stage similarity-join based framework. In the first stage, for each column, we identify the types which share at least one common entity/cell value with the column. In the second stage, we count the number of their shared entities/cell values and retrieve top k types with the largest similarity. Algorithm 1 illustrates the pseudo-code.

Stage 1: For each column, find the list of types that share at least one entity/cell value with the column.

`Map`: $\langle \mathtt{C}, \{v_1, v_2, \ldots\}\rangle \rightarrow \langle v_i, \mathtt{C}\rangle$. $\langle \mathtt{T}, \{e_1, e_2, \ldots\}\rangle \rightarrow \langle e_i, \mathtt{T}\rangle$.

It takes as input a sequence of columns (each column includes a set of cell values) from Web tables and a set of types (each type includes a set of entities) from a knowledge base. For each column `C`, it outputs key-value pairs $\langle v_i, \mathtt{C}\rangle$ where v_i is any cell value of `C`. For each type `T`, it outputs key-value pairs $\langle e_i, \mathtt{T}\rangle$ where e_i is any entity of `T`.

`Reduce`: $\langle v_i = e_i, list(\mathtt{C/T})\rangle \rightarrow \langle \mathtt{C}, list(\mathtt{T})\rangle$.

It takes as input a set of key-value pairs $\langle v_i = e_i, list(\mathtt{C/T})\rangle$, where $v_i = e_i$ is a cell value/an entity, and $list(\mathtt{C/T})$ is a list of columns/types that contain the cell value/entity. The `Reducer` separates $list(\mathtt{C/T})$ into a list of columns, denoted by $list(\mathtt{C})$ and a list of types, denoted by $list(\mathtt{T})$. (We use a flag to differentiate `C/T` to be a column or a type.) For each column `C` in $list(\mathtt{C})$, it outputs a key-value pair $\langle \mathtt{C}, list(\mathtt{T})\rangle$.

`Combine`: We can use `Combine` to do a local reduce. As it is not a focus of this paper, we will not discuss the details.

Stage 2: Compute top-k types of every column.

`Map`: $\langle \mathtt{C}, list(\mathtt{T})\rangle \rightarrow \langle \mathtt{C}, list(\mathtt{T})\rangle$.

It takes as input key-value pairs $\langle \mathtt{C}, list(\mathtt{T})\rangle$, where $list(\mathtt{T})$ is a list of types that share an entity/cell value with `C`. It outputs the same key-value pairs as input.

`Reduce`: $\langle \mathtt{C}, list(list(\mathtt{T}))\rangle \rightarrow \langle \mathtt{C}, \mathtt{topk(T)}\rangle$.

It takes as input key-value pairs $\langle \mathtt{C}, list(list(\mathtt{T}))\rangle$, where $list(list(\mathtt{T}))$ is a list of lists of types that share common entities/cell values with `C`. We use a hash based method to compute the overlap similarity of `C` and a type `T` in $list(list(\mathtt{T}))$. We first initialize a hash table. Then for each type in $list(list(\mathtt{T}))$, if it is not in the hash table, we add it into the hash table and set the occurrence number as 1; otherwise we increase the number by 1. After scanning all types, the occurrence number of a type is exactly the overlap similarity of the type with the column. Based on the number, we rank the types and add the top-k types with the largest overlap similarity into `topk(T)`. Finally it outputs key-value pairs $\langle \mathtt{C}, \mathtt{topk(T)}\rangle$.

Algorithm 1: Similarity-Join Based Framework

```
   // the first stage
1  Map(⟨C/T, {v1, v2, ...}/{e1, e2, ...}⟩)
2      foreach column ⟨C, {v1, v2, ...}⟩ do output(⟨vi, C⟩);
3      foreach type ⟨T, {e1, e2, ...}⟩ do output(⟨ei, T⟩);
4  Reduce (⟨vi = ei, list(C/T)⟩)
5      foreach C/T in list(C/T) do
6          if C/T is a type then add T to list(T);
7          else if C/T is a column then add C to list(C);
8      foreach C in list(C) do output(⟨C, list(T)⟩);
   // the second stage
9  Map (⟨C, list(T)⟩)
10     output(⟨C, list(T)⟩);
11 Reduce (⟨C, list(list(T))⟩)
12     Initialize a hash map H ;
13     foreach T in list(list(T)) do
14         if T ∈ H then H(T) = H(T) + 1;
15         else H(T) = 1;
16     topk(T) = top-k types with largest H(T);
17     output(⟨C, topk(T)⟩);
```

Example 1. *Figure 2 shows four columns and four types. Figure 3 illustrates a running example that uses our framework to label each column. In the first stage, the* `Map` *step reads the column file and the type file. For each column (type), it outputs ⟨cell value, column⟩(⟨entity, type⟩) pairs. For* $\mathtt{C}_1$*, it outputs six key-value pairs* $\langle v_1, \mathtt{C}_1\rangle, \ldots, \langle v_6, \mathtt{C}_1\rangle$*. For* $\mathtt{T}_2$*, it outputs two key-value pairs* $\langle e_1, \mathtt{T}_2\rangle, \langle e_2, \mathtt{T}_2\rangle$*. In the* `Reduce` *step, it reads the key-value pairs grouped by the key(cell value or entity). Consider* $\langle v_1 = e_1, \{\mathtt{C}_1, \mathtt{C}_2, \mathtt{T}_1, \mathtt{T}_2\}\rangle$*.* $\mathtt{T}_1, \mathtt{T}_2$ *share an entity* e_1 *with* $\mathtt{C}_1, \mathtt{C}_2$*. It outputs* $\langle \mathtt{C}_1, \{\mathtt{T}_1, \mathtt{T}_2\}\rangle$ *and* $\langle \mathtt{C}_2, \{\mathtt{T}_1, \mathtt{T}_2\}\rangle$*. In the second stage, the* `Map` *step outputs the same key-value pairs as the input. The* `Reduce` *step computes top-k types for each column by counting the occurrence number of each type in the key-value pair. Consider* $\langle \mathtt{C}_1, \{\{\mathtt{T}_1, \mathtt{T}_2\}, \{\mathtt{T}_1, \mathtt{T}_2\}, \{\mathtt{T}_1, \mathtt{T}_3\}, \{\mathtt{T}_1, \mathtt{T}_3\}\}\rangle$*.* $\mathtt{T}_1$ *appears four times, and* $\mathtt{T}_2$ *and* $\mathtt{T}_3$ *appear twice. Thus the top-1 type for* $\mathtt{C}_1$ *is* $\mathtt{T}_1$*. Similarity the top-1 type for* $\mathtt{C}_2$ *(*$\mathtt{C}_3$*) is* $\mathtt{T}_2$ *(*$\mathtt{T}_3$*). Notice that we cannot label* $\mathtt{C}_4$ *since it does not share any entity with any type.*

4.2 Extension to Other Functions

Exact-matching Similarity Functions: Different from the overlap similarity, besides $|\mathtt{V_C} \cap \mathtt{E_T}|$, other exact-matching functions, e.g., weighted Jaccard, also rely on $|\mathtt{V_C}|$, $|\mathtt{E_T}|$, and the weight of each entity/cell value. To support these functions, we add these parameters into the key-value pairs, and in the `Reduce` step of the second stage, we compute the sim-

Column C_1	
v_1	Forrest Gump
v_2	Catch Me If You Can
v_3	Saving Private Ryan
v_4	Schindler's List
v_5	Shark Night 3D
v_6	Da Vinci Code
v_7	The Terminal
v_8	Full Metal Jacket

Column C_2	
v_1	Forrest Gump
v_2	Catch Me If You Can
v_7	The Terminal

Column C_3	
v_3	Saving Private Ryan
v_4	Schindler's List
v_8	Full Metal Jacket

Column C_4	
v_5	Shark Night 3D
v_6	Da Vinci Code

Type T_1 (/film)	
e_1	Forrest Gump
e_2	Catch Me If You Can
e_3	Saving Private Ryan
e_4	Schindler's List
e_5	Shark Night
e_6	The Da Vinci Code
e_7	The Green Mile
e_8	Heartbreak Ridge

Type T_2 (/film/drama)	
e_1	Forrest Gump
e_2	Catch Me If You Can
e_7	The Green Mile

Type T_3 (/film/war)	
e_3	Saving Private Ryan
e_4	Schindler's List
e_8	Heartbreak Ridge

Type T_4 (/film/thriller)	
e_5	Shark Night
e_6	The Da Vinci Code

Figure 2: Example columns and types.

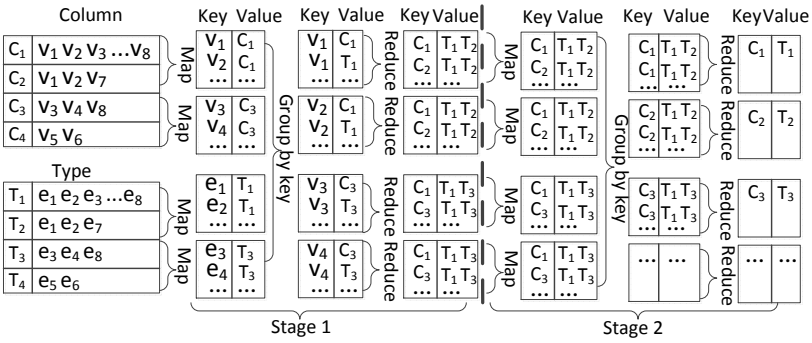

Figure 3: Running example of our framework ($v_1 = e_1$, $v_2 = e_2$, $v_3 = e_3$, $v_4 = e_4$. $v_5 \sim e_5$, $v_6 \sim e_6$ ($\sim$: similar)).

ilarity based on these parameters. Thus we can extend this framework to support other exact-matching functions.

Fuzzy-matching Similarity Functions: To support fuzzy-matching functions, we slightly modify the framework. For each cell value (or entity), we generate its signatures. A cell value and an entity is similar only if they share at least one common *signature*. There are many signature strategies, e.g., q-gram [11] and prefix signature [28]. We can use any of them in our method. Next we modify the basic framework to support fuzzy-matching functions based on the signatures.

Stage 1: For each column, find the list of types that share at least one common *signature* with the column.

Map: $\langle \mathtt{C}, \{v_1, \ldots\}\rangle \rightarrow \langle \mathtt{sig}, \langle \mathtt{C}, v_i\rangle\rangle$. $\langle \mathtt{T}, \{e_1, \ldots\}\rangle \rightarrow \langle \mathtt{sig}, \langle \mathtt{T}, e_j\rangle\rangle$.

For each column $\mathtt{C}$, for each cell value v_i, it computes the signatures and for each signature $\mathtt{sig}$, generates key-value pairs, $\langle \mathtt{sig}, \langle \mathtt{C}, v_i\rangle\rangle$. For each type $\mathtt{T}$, for each entity e_j, it computes the signatures and for each signature $\mathtt{sig}$, generates key-value pairs $\langle \mathtt{sig}, \langle \mathtt{T}, e_j\rangle\rangle$.

Reduce: $\langle \mathtt{sig}, list(\langle \mathtt{C}, v_i\rangle / \langle \mathtt{T}, e_j\rangle)\rangle \rightarrow \langle \mathtt{C}, list(\langle \mathtt{T}, v_i, e_j\rangle)\rangle$.

It separates the list into the column list and the type list. For each column, for each type, if the column has a cell value that shares a common signature with an entity of the type, it adds the type into a type list $list(\langle \mathtt{T}, v_i, e_j\rangle)$. Finally it outputs key-value pairs $\langle \mathtt{C}, list(\langle \mathtt{T}, v_i, e_j\rangle)\rangle$.

Stage 2: Compute top-k types of every column.

Map: $\langle \mathtt{C}, list(\langle \mathtt{T}, v_i, e_j\rangle)\rangle \rightarrow \langle \mathtt{C}, list(\langle \mathtt{T}, v_i, e_j\rangle)\rangle$.

It outputs the same key-value pairs as the input.

Reduce: $\langle \mathtt{C}, list(list(\langle \mathtt{T}, v_i, e_j\rangle))\rangle \rightarrow \langle \mathtt{C}, \mathtt{topk(T)}\rangle$.

For each column, it gets a list of lists of types. Then it computes the fuzzy-matching similarity using existing algorithms [29]. Based on the fuzzy-matching similarity, we rank the types and identify the top-k types with the largest fuzzy-matching similarity to annotate the column.

EXAMPLE 2. *Consider column* $\mathtt{C}_4$ *and type* $\mathtt{T}_4$. *Suppose the signatures of* v_5 = *"Shark Night 3D" and* e_5 = *"Shark Night" are* $\mathtt{sig}_1$ = *"Shark". The signatures of* v_6 = *"Da Vinci Code" and* e_6 = *"The Da Vinci Code" are* $\mathtt{sig}_2$ = {*"Vinci"*}. *In the* Map *step of the first stage, we generate pairs* $\langle \mathtt{sig}_1, \langle \mathtt{C}_4, v_5\rangle\rangle$ *and* $\langle \mathtt{sig}_2, \langle \mathtt{C}_4, v_6\rangle\rangle$ *for* $\mathtt{C}_4$ *and* $\langle \mathtt{sig}_1, \langle \mathtt{T}_4, v_5\rangle\rangle$ *and* $\langle \mathtt{sig}_2, \langle \mathtt{T}_4, v_6\rangle\rangle$ *for* $\mathtt{T}_4$. *In the* Reduce *step, we generate* $\langle \mathtt{C}_4, \langle \mathtt{T}_4, v_5, e_5\rangle\rangle$ *and* $\langle \mathtt{C}_4, \langle \mathtt{T}_4, v_6, e_6\rangle\rangle$. *In the* Reduce *step of the second stage, we compute the fuzzy similarity. The top-1 type of* $\mathtt{C}_4$ *is* $\mathtt{T}_4$. *Thus the fuzzy matching functions can annotate* $\mathtt{C}_4$ *by tolerating inconsistencies between entities and cell values.*

4.3 Observations

Optimization Goal: To improve the performance, our optimization goal is to reduce the number of key-value pairs. Although we can use the prefix-filtering technique [28], it has the following problems. First, it depends on a given threshold and cannot support our top-k problem. Second, it involves other expensive overhead[28], e.g., pre-defining a global order and selecting high-quality prefixes based on the global order. To address these problems, we propose effective pruning technique based on the following observations.

Observation 1: We can aggregate the types into different groups. For each entity, to generate the key-value pairs, we use a group to replace the types in the group. Since each group contains multiple types, this method can reduce the number of key-value pairs. In addition, a group may be used by multiple entities, and we can share the computations for the types in the same group. We will discuss how to utilize groups to label columns in Section 5.

Observation 2: We can partition the cell values into different partitions. For each column, we utilize the partitions to generate its key-value pairs. Since each partition contains multiple cell values, this method can reduce the number of key-value pairs. Moreover, if the cell values of a column fall into one partition, we can utilize the partition to directly annotate the column and prune other unnecessary partitions. We will discuss the details in Section 6.

In the following sections, we focus on improving the efficiency and scalability. For simplicity, we take overlap similarity as an example, and our techniques can be easily applied to other similarity functions.

5. KNOWLEDGE TYPE AGGREGATION

For each entity, we want to reduce the number of key-value pairs produced for the entity. Let $\mathcal{L}(e)$ denote the list of types that contain entity e. We aggregate types in $\mathcal{L}(e)$ and generate several disjoint groups $\mathcal{G}_1(e)$, $\mathcal{G}_2(e)$, ..., $\mathcal{G}_l(e)$. Instead of producing key-value pairs $\langle e, \mathtt{T}\rangle$ for every type $\mathtt{T} \in$

$L(e)$, we generate key-value pairs $\langle e; G_i(e)\rangle$ for every group $G_i(e)$ for $1 \le i \le l$. That is we use group $G_i(e)$ to replace (multiple) types in the group. Obviously this method can reduce the number of key-value pairs and can improve the performance. There are several research challenges in this type aggregation based method. The first one is how to utilize the groups to compute top-k types for each column. We formally introduce an aggregation framework to address this issue in Section 5.1. The second challenge is to quantify different aggregation strategies. We propose a cost based model to analyze the group quality in Section 5.2. The third one is to efficiently generate high-quality groups. We devise efficient algorithms to address this challenge in Section 5.3.

5.1 Aggregation Framework

Given a knowledge base, for each entity e, we aggregate the types in its inverted list $L(e)$ and partition them into l disjoint groups $G_1(e)$, $G_2(e)$, $\ldots$, $G_l(e)$, such that

(1) Subset: $G_i(e)$ is a subset of $L(e)$ for $1 \le i \le l$;

(2) Completeness: $\bigcup_{1 \le i \le l} G_i(e) = L(e)$; and

(3) Disjoint: $G_i(e) \cap G_j(e) = \emptyset$ for $i \ne j$.

Let $G_T = \{G_1, G_2, \ldots, G_g\}$ denote the set of distinct groups for all entities. Let H_T^G denote a hash table from the distinct groups to lists of types, i.e., each group G_i is associated with a list of types contained in G_i, $H_T^G(G_i)$.

For example, consider the types and entities in Figure 2. $L(e_1) = \{T_1, T_2\}$. $L(e_2) = \{T_1, T_2\}$. $L(e_3) = \{T_1, T_3\}$. $L(e_4) = \{T_1, T_3\}$. $L(e_5) = \{T_1, T_4\}$. $L(e_6) = \{T_1, T_4\}$. $L(e_7) = \{T_1, T_2\}$. $L(e_8) = \{T_1, T_3\}$. The basic framework will generate 16 key-value pairs. Suppose we aggregate them into three groups, $G_1 = \{T_1, T_2\}; G_2 = \{T_1, T_3\}; G_3 = \{T_1, T_4\}$. We generate 8 key-value pairs $\langle e_1; G_1\rangle$, $\langle e_2; G_1\rangle$, $\langle e_3; G_2\rangle$, $\langle e_4; G_2\rangle$, $\langle e_5; G_3\rangle$, $\langle e_6; G_3\rangle$, $\langle e_7; G_1\rangle$, and $\langle e_8; G_2\rangle$.

Suppose we can get the groups of each entity, i.e., $\langle e; \{G_1(e), G_2(e), \ldots, G_l(e)\}\rangle$. We will discuss how to generate the high-quality groups in Section 5.3. Next we discuss how to use the groups to compute top-k types of each column. We still employ a two-stage MapReduce framework. Algorithm 2 illustrates the pseudo-code.

Stage 1: For each column, find the list of types that share at least one entity/cell value with the column.

Map $\langle C; \{v_1, v_2, \ldots\}\rangle \to \langle v_i; C\rangle$. $\langle e; \{G_1(e), \ldots, G_l(e)\}\rangle \to \langle e; G_i(e)\rangle$.

The input and output for the columns are the same as those of the basic framework. For the types, the input is a set of key-value pairs $\langle e; \{G_1(e), G_2(e), \ldots, G_l(e)\}\rangle$. For each entity, it outputs key-value pairs $\langle e; G_i(e)\rangle$ for $1 \le i \le l$.

Reduce $\langle v = e; \text{list}(C/G)\rangle \to \langle C; \text{list}(G)\rangle$.

Different from the basic framework, for each column, it outputs key-value pairs $\langle C; \text{list}(G)\rangle$, where G is a group which has a type sharing a common entity with C.

Stage 2: Compute top-k types of every column.

Map $\langle C; \text{list}(G)\rangle \to \langle C; \text{list}(G)\rangle$.

Similarly it outputs key-value pairs $\langle C; \text{list}(G)\rangle$.

Reduce $\langle C; \text{list}(\text{list}(G))\rangle \to \langle C; \text{topk}(T)\rangle$.

For each column C, it gets a list of lists of groups. We still use a hash based method to compute the overlap similarity of C and a type $T \in G \in \text{list}(\text{list}(G))$. We first initialize a hash table. Then for each group G in $\text{list}(\text{list}(G))$, we get the list of types in the group, i.e., $H_T^G(G)$. For each type in $H_T^G(G)$, we compute its occurrence number similar as the basic framework. Finally we identify top-k types with the

Algorithm 2: Aggregation-based Algorithm

```
   // the first stage
1  Map(⟨C=e; {v1; v2; :::g={G1(e); G2(e); :::gi)
2  |  foreach column ⟨C; {v1; v2; :::gi do output(⟨vi; Ci);
3  |  foreach entity ⟨e; {G1(e); G2(e); :::gi do
   |  output(⟨e; Gi(e)i);
4  Reduce(⟨v = e; list(C=G)i)
5  |  foreach C=T in list(C=G) do
6  |  |  if C=T is a group then add G to list(G);
7  |  |  else if
```

(a) Variable gadget $\mathcal{X}^g$ (b) Clause gadget $\mathcal{C}^g$

Figure 14: Variable and Clause Gadgets. Each vertex represents a distinct type and each hyperedge represents an entity.

Figure 15: An example of reducing 3-CNF to finding aggregation strategy problem.

PROOF. First, given any aggregation strategy, we can easily check whether the cost is no larger than t in polynomial time.

Next we prove the decision problem is NPC by a reduction from the 3-CNF-SAT problem. In particular, given any instance ϕ of a 3-CNF-SAT problem, we construct an instance of our decision problem $\langle \mathcal{H}, t \rangle$ such that the 3-CNF formula ϕ is satisfiable if and only if there exists an aggregation strategy of $\mathcal{H}$ with cost no larger that t. In the reduction we let $t = 16n + 24k$ where n and k are respectively the numbers of variables and clauses in ϕ. The construct of $\mathcal{H}$ is as follows (an example can be found in Figure 15). For each variable x_i in ϕ, we create a variable gadget $\mathcal{X}_i^g$ which consists of three entities $e_1^i = \{T_1^i, T_2^i, T_3^i, T_4^i, T_5^i, T_8^i\}$, $e_2^i = \{T_2^i, T_3^i, T_6^i, T_8^i\}$ and $e_3^i = \{T_4^i, T_5^i, T_7^i, T_8^i\}$. Recall that the tip T_6^i is labeled with the literal x_i and tip T_7^i is labeled with literal $\overline{x}_i$. For each clause C_j in ϕ, we create a clause gadget $\mathcal{C}_j^g$ which consists of seven entities $e_4^j = \{T_9^j, T_{10}^j, T_{11}^j\}$, $e_5^j = \{T_{11}^j, T_{13}^j, T_{14}^j\}$, $e_6^j = \{T_{10}^j, T_{12}^j, T_{13}^j\}$, $e_7^j = \{T_{10}^j, T_{11}^j, T_{13}^j\}$, $e_8^j = \{T_{10}^j, T_{l_1}^j\}$, $e_9^j = \{T_{11}^j, T_{l_2}^j\}$ and $e_{10}^j = \{T_{13}^j, T_{l_3}^j\}$. Recall that the tips $T_{l_1}^j$, $T_{l_2}^j$ and $T_{l_3}^j$ are labeled with the three literals of clause C_j. The variable gadgets and clause gadgets are connected as follows. We identify together each tip in $\mathcal{C}_j^g$ with the tip in the variable gadget that has the same label (i.e., they corresponds to the same literal). This finishes the construction of $\mathcal{H}$.

The reduction creates 3 entities and 8 types for each variable x_i and 7 entities and 6 types for each clause C_j. We can perform the reduction algorithm which totally creates $3n + 7k$ entities and $8n + 6k$ types in polynomial time as it produces each entity or type in the constant time.

We next show that the 3-CNF formula ϕ is satisfiable if and only if there exists an aggregation strategy for $\mathcal{H}$ with cost no larger than $t = 16n + 24k$.

First, suppose there is a truth assignment that satisfies ϕ. We prove there must exist an aggregation strategy with cost no larger than $t = 16n + 24k$ as follows. For each $1 \leq i \leq n$, if $x_i = \mathsf{True}$, we aggregate variable gadget $\mathcal{X}_i^g$ using aggregation strategy $\mathcal{S}_{V_1}$ with cost 16 and $G_x^i = \langle T_6^i \rangle$. If $x_i = \mathsf{False}$, we aggregate variable gadget $\mathcal{X}_i^g$ using aggregation strategy $\mathcal{S}_{V_2}$ with cost 16 and $G_x^i = \langle T_7^i \rangle$. By using this aggregation strategy to aggregate every variable gadget $\mathcal{X}_i^g (1 \leq i \leq n)$, the total cost is $16n$. For each $1 \leq j \leq k$, as ϕ is satisfied, C_j contains at least one literal $l_m (1 \leq m \leq 3)$ whose value is True. Suppose l_m is the first literal in C_j whose value is evaluated to be True in the truth assignment. We aggregate clause gadget $\mathcal{C}_j^g$ using aggregation strategy $\mathcal{S}_{L_m}$ with cost 24 and $G_C^j = \langle T_{l_m}^j \rangle$. Moreover, we can see that the group G_C^j also exists in the aggregation strategy chosen for variable gadget $\mathcal{X}_i^g$ since l_m is True (more specifically, $G_C^j = G_x^i$). Hence, the additional cost $c_j = 0$ (the cost of $|G_C^j|$ has been already counted in the variable clause). Using this strategy, the total cost to aggregate all $\mathcal{C}_j^g$ is $24k$ and the additional cost is 0. Based on the discussion above, the overall cost to aggregate entity set $\mathcal{H}$ is $16n + 24k$.

Next suppose that there is an aggregation strategy of $\mathcal{H}$ with cost no larger than $16n + 24k$. We prove there must exist an assignment that satisfies ϕ. For any variable gadget $\mathcal{X}_i^g (1 \leq i \leq n)$, the minimum aggregation cost is 16. Thus the minimum cost to aggregate all variable gadgets is $16n$. For any clause gadget $\mathcal{C}_j^g (1 \leq j \leq k)$, the minimum aggregation cost is 24. Thus the minimum cost to aggregate all clause gadgets is $24k$. Thus to achieve an aggregation cost of $\mathcal{H}$ no larger than $16n+24k$, we need to apply the aggregation strategy with minimum cost to all the variable gadgets and clause gadgets and all the additional costs $c_j (1 \leq j \leq k)$ should be 0. According to Property 1 and Property 2, there must exist a group G_x^i in each variable gadget $\mathcal{X}_i^g$ which only contains one of the two tips labeled with x_i and $\overline{x}_i$ and there must exist a group G_C^j in each clause gadgets $\mathcal{C}_j^g$ which only contains one of the three tips labeled with the literals. To make all the additional cost $c_j = 0$, each group G_C^j should also exist in its corresponding variable gadgets $\mathcal{X}_i^g$ which is exactly G_x^i. We set $x_i = \mathsf{True}$ if $G_x^i = G_C^j$ contains only the tip labeled with x_i. Otherwise we set $x_i = \mathsf{False}$ if $G_x^i = G_C^j$ contains only the tip labeled with $\overline{x}_i$. By this assignment, any clause $C_j (1 \leq j \leq k)$ is satisfied as C_j contains a literal x_i when $x_i = \mathsf{True}$ (as G_C^j contains the tip labeled with x_i) and contains literal $\overline{x}_i$ when $x_i = \mathsf{False}$ (as G_C^j contains the tip labeled with $\overline{x}_i$). Note that this assignment will not set $x_i = 1$ and $x_i = 0$ simultaneously. This is because based on Property 1, any aggregation of a variable gadget of cost 16 contains exactly one singleton group which contain exactly one tip. Thus the Theorem 4 is proved. $\square$