Scalable Column Concept Determination for Web Tables Using Large Knowledge Bases

Dong Deng[†] Yu Jiang[†] Guoliang Li[†] Jian Li[‡] Cong Yu[#]
[†]Department of Computer Science, Tsinghua University, Beijing, China.
[‡]Institute for Interdisciplinary Information Sciences,Tsinghua University, Beijing, China.
[#]Google Research, New York, USA
dd11@mails.thu.edu.cn, sunlight07@126.com, {liguoliang,lijian83}@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 effective 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 efficient 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 finding the optimal aggregation strategy and that of finding the optimal partition strategy are NP-hard, and propose efficient heuristic techniques by leveraging the *hierarchy* of the knowledge base. Experimental results on real-world datasets show that our method achieves high annotation quality and performance, and scales well.

1. INTRODUCTION

The Web contains a vast amount of structured data in the form of tables. According to a recent Google study [6], there are a total of 14 billion raw HTML tables and, among those, 150 million relational data tables, easily making them one of the richest structured data sources. As a result, tables on the Web have received significant interests from both industry [27, 34] and academia [18], with the focus on understanding the semantics of the tables. More recently, those tabular data have begun to enrich users' search experience [13, 35]. For example, Google's Table Search* are helping users browsing and finding structured data through queries such as "asian country gdp."

To leverage Web tables for applications such as table search and data integration, a number of tasks must be performed on the tables to recover their semantics. Those tasks include, but are not limited to, annotating cell values (e.g., matching the cell's textual content to an entity in a knowledge base), discovering table subjects, and identifying table schema. None of those tasks are easy because of the heterogeneous nature of data on the Web. Underneath all those tasks, one of the core table understanding tasks is *column concept determination*, i.e., given a column of cells within a Web table, determining the most likely concept for the column, where a concept is a type (or class) within a given knowledge base. For example, a better understanding of the dominant concept(s) of a column can help constrain cell-level annotations for cells within the column, leading to better cell annotations.

Not surprisingly, due to its importance, column concept determination is studied by both [18] and [27]. In [18], column concepts are discovered as part of a graphical model that is employed to collectively learn three things at the same time: column concepts, binary relationships between multiple columns, and class labels on cells. Intuitively, this approach achieves better quality, but is in general not very scalable due to the expensive learning model being employed. In [27], column concepts are determined using a simpler majority-rule mechanism of selecting the concept(s) that the most cells in the column have been mapped to, but with a much larger knowledge base that is derived from the Web. There are several drawbacks of this approach. First, the majority-rule mechanism can lead to mistakes because of the presence of *stop* concepts, i.e., concepts such as "`music`" that can often be mapped to many textual contents indiscriminately. Second, only flat knowledge bases are considered, leaving the semantics of knowledge hierarchies underutilized. Third, only exact matching is used to map cell values to the knowledge base, and extending exact matching to approximate matching is a non-trivial task that is made more challenging by the ever increasing size of the available knowledge bases [1, 10, 25, 31].

Similar to [27], we focus on the same important instantiation of the column concept determination problem, i.e., identifying a column's top k concepts based on matching its content to a large knowledge base. However, instead

VLDB '13, August 26-31, 2013, Riva del Garda, Italy
Copyright 2013 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

*`http://research.google.com/tables`

of restricting the matching to be exact and the ranking of the concept to be majority-rule (which are what made the previous approach scalable), we aim to provide a solution that allows both *fuzzy matching* and ranking of the concepts based on *more general similarity functions*, without sacrificing scalability. This solution enables us to produce column concepts that are as good as those produced by [18], but in a much more scalable fashion.

More specifically, we identify top k similar concepts from the knowledge base, where the similarity between a concept and a column is quantified based on the matching between the entities in the concept and the cell values of the column. Fuzzy matching is enabled between the cell values and knowledge entities, such that "`George Bush`" and "`George W. Bush`" can be matched. A more general set of similarity functions can be adopted to allow more flexible ranking of the candidate concepts than simple majority-rule. Both features are accomplished without compromising the scalability of the whole process.

Contributions: (1) We extend MapReduce-based similarity-joins to support scalable column concept determination. (2) We propose a knowledge concept aggregation based method which aggregates knowledge concepts into groups. Computations can then be shared for concepts in the same group. (3) We devise a knowledge entity partition based method which partitions entities into different partitions. For each column, only the necessary partitions are used to annotate the column while large numbers of unnecessary partitions are pruned. (4) We prove that both the problem of finding the optimal aggregation strategy and that of finding the optimal partition strategy are NP-hard. We utilize the knowledge hierarchy to heuristically generate high-quality groups and partitions. (5) Experimental results on real-world datasets show that our method achieves high annotation quantity and performance, and scales well.

Paper Structure: We formalize the problem in Section 2 and review related works in Section 3. A similarity-join based framework is discussed in Section 4. We propose an aggregation based method in Section 5 and a partition based method in Section 6. Experimental results are reported in Section 7. We conclude the paper in Section 8.

2. PRELIMINARIES

2.1 Problem Statement

Data Model: A knowledge base consists of many types (or concepts). In the paper we use the terms "concepts" and "types" interchangeably. Each type has a set of large numbers of entities (or instances). The knowledge base has a hierarchy and can be modelled as a Directed Acyclic Graph. Each node in the graph is a type or an entity. Each edge between two type nodes indicates the subtype relationship of the two types. Each edge between a type node and an entity node denotes the entity belongs to the type. For example, Figure 1 shows a partial knowledge base, where the leaf-level nodes are entities, e.g., "Forrest Gump", and intermediate-level nodes are types, e.g., "`/film`" and "`/film/drama`". Type "`/film/drama`" is a subtype of "`/film`". "Forrest Gump" and "Catch Me If You Can" are entities of type "`/film/drama`".

We consider a table as consisting of a sequence of columns, each of which has a set of cell values. For example, Table 1 illustrates a Web table with 4 columns.

Problem Formulation: Given a large set (tens or hundreds of millions) of tables from the Web, and a comprehensive knowledge base (with tens or hundreds of millions of entities), for each column of all the tables, efficiently identify top-k types from the knowledge base that can be used to describe the column based on a user provided similarity function. More specifically, a column `C` can be described by a type `T` if `T` shares *significant similarity* with `C`. Our goal is to identify top-k types with the highest similarities to label all the columns, with an emphasis on efficiency and scalability. In the paper we focus on using the cell content and extracting concepts from the content of the cells. For example, the second column of the table in Table 1 should be annotated with "`/film`", while the third column should be annotated with "`/people/director`".

To better describe a column, the identified types and the column should have large similarity. Based on this observation, we utilize similarity functions to quantify the similarity between a type and a column.

2.2 Similarity Functions

Given a column `C` and a type `T`, we define their similarity as the set similarity of `C`'s cell-value set, denoted by $\mathtt{V_C}$, and `T`'s entity set, denoted by $\mathtt{E_T}$. Existing set similarity can be broadly classified into two categories: exact matching similarity and fuzzy matching similarity.

Exact-matching Similarity Functions: They define the similarity based on the overlap of the two sets. We take the following well-known set similarity functions as examples.

Overlap Similarity: $\textsc{Sim}(\mathtt{C},\mathtt{T}) = |\mathtt{V_C} \cap \mathtt{E_T}|$.

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

Cosine Similarity: $\textsc{Sim}(\mathtt{C},\mathtt{T}) = \frac{|\mathtt{V_C} \cap \mathtt{E_T}|}{\sqrt{|\mathtt{V_C}||\mathtt{E_T}|}}$.

Dice Similarity: $\textsc{Sim}(\mathtt{C},\mathtt{T}) = \frac{2|\mathtt{V_C} \cap \mathtt{E_T}|}{|\mathtt{V_C}|+|\mathtt{E_T}|}$.

Weighted Jaccard Similarity: $\textsc{Sim}(\mathtt{C},\mathtt{T}) = \frac{\sum_{t \in \mathtt{V_C} \cap \mathtt{E_T}} w_t}{\sum_{t \in \mathtt{V_C} \cup \mathtt{E_T}} w_t}$,

where w_t is the weight of a cell value or an entity.

Fuzzy-matching Similarity Functions: Exact matching similarity functions can be restricted as they cannot capture fuzzy matching between cell values and entities. However in many real-world applications, we need to accommodate fuzzy matching [7]. For example, many Web tables contain typing errors and an entity has different representations. Furthermore, some columns may match multiple types, e.g., movie and director. Thus we need to enable fuzzy matching between an entity and a cell value.

Before we introduce fuzzy-matching similarity functions, we first define some notations. Let $\textsc{eSim}(v, e)$ denote the similarity between a cell value v and an entity e. We can use existing similarity functions to quantify their similarity, e.g., Jaccard, Cosine, and Edit Distance[†]. Given a type `T` and a column `C`, a set of ⟨cell value, entity⟩ pairs, denoted by $\mathcal{M} = \{(v_i, e_j) | v_i \in \mathtt{V_C}, e_j \in \mathtt{E_T}\}$, is a matching if each v_i and e_j appears at most once in $\mathcal{M}$. The maximum matching $\mathcal{M}_{max}$ is the matching with the maximum value $\sum_{(v_i,e_j) \in \mathcal{M}_{max}} \textsc{eSim}(v_i, e_j)$. We can use existing algorithms [29] to compute the maximum matching. We use $\mathtt{V_C} \widetilde{\cap} \mathtt{E_T}$

[†] The edit distance between two elements is the minimum number of edit operations (insertion, deletion, substitution) needed to transform one element to another.

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, ...

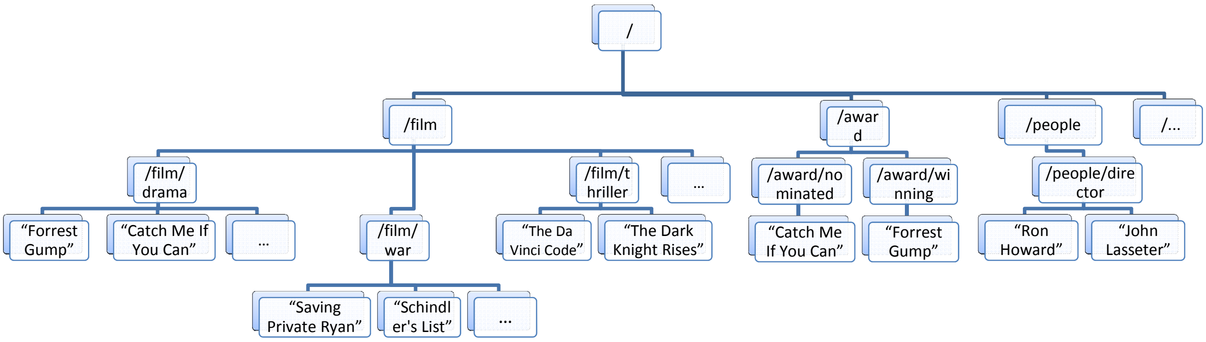


Figure 1: Example knowledge base based on Wikipedia.

to denote the maximum matching, called fuzzy overlap of $\mathtt{T}$ and $\mathtt{C}$, and $|\mathtt{V}_\mathtt{C} \tilde{\cap} \mathtt{E}_\mathtt{T}|$ to denote its value.

We can extend exact-matching functions by replacing $\mathtt{V}_\mathtt{C} \cap \mathtt{E}_\mathtt{T}$ with $\mathtt{V}_\mathtt{C} \tilde{\cap} \mathtt{E}_\mathtt{T}$ to define fuzzy-matching functions. Next we take fuzzy Jaccard similarity as an example.

Fuzzy Jaccard Similarity: $\textsc{Sim}(\mathtt{C}, \mathtt{T}) = \frac{|\mathtt{V}_\mathtt{C} \tilde{\cap} \mathtt{E}_\mathtt{T}|}{|\mathtt{V}_\mathtt{C}| + |\mathtt{E}_\mathtt{T}| - |\mathtt{V}_\mathtt{C} \tilde{\cap} \mathtt{E}_\mathtt{T}|}$.

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

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

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

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

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

It takes as input key-value pairs $\langle \mathtt{C}, \mathit{list}(\mathit{list}(\mathtt{T}))\rangle$, where $\mathit{list}(\mathit{list}(\mathtt{T}))$ 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 $\mathit{list}(\mathit{list}(\mathtt{T}))$. We first initialize a hash table. Then for each type in $\mathit{list}(\mathit{list}(\mathtt{T}))$, 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 $\langle \mathtt{C}, \mathtt{topk}(\mathtt{T})\rangle$.

Algorithm 1: Similarity-Join Based Framework

```
// the first stage
Map($\langle \mathtt{C/T}, \{v_1, v_2, \ldots\}/\{e_1, e_2, \ldots\}\rangle$)
    foreach column $\langle \mathtt{C}, \{v_1, v_2, \ldots\}\rangle$ do output($\langle v_i, \mathtt{C}\rangle$);
    foreach type $\langle \mathtt{T}, \{e_1, e_2, \ldots\}\rangle$ do output($\langle e_i, \mathtt{T}\rangle$);
Reduce ($\langle v_i = e_i, \mathit{list}(\mathtt{C/T})\rangle$)
    foreach C/T in list(C/T) do
        if C/T is a type then add T to list(T);
        else if C/T is a column then add C to list(C);
    foreach C in list(C) do output($\langle \mathtt{C}, \mathit{list}(\mathtt{T})\rangle$);
// the second stage
Map ($\langle \mathtt{C}, \mathit{list}(\mathtt{T})\rangle$)
    output($\langle \mathtt{C}, \mathit{list}(\mathtt{T})\rangle$);
Reduce ($\langle \mathtt{C}, \mathit{list}(\mathit{list}(\mathtt{T}))\rangle$)
    Initialize a hash map $\mathcal{H}$ ;
    foreach T in list(list(T)) do
        if $\mathtt{T} \in \mathcal{H}$ then $\mathcal{H}(\mathtt{T}) = \mathcal{H}(\mathtt{T}) + 1$;
        else $\mathcal{H}(\mathtt{T}) = 1$;
    topk(T) = top-$k$ types with largest $\mathcal{H}(\mathtt{T})$;
    output($\langle \mathtt{C}, \mathtt{topk}(\mathtt{T})\rangle$);
```

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* C_2 *(*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

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

Column C$_2$	
v_1	Forrest Gump
v_2	Catch Me If You Can
v_7	The Terminal

Type T$_2$ (/film/drama)	
e_1	Forrest Gump
e_2	Catch Me If You Can
e_7	The Green Mile

Column C$_3$	
v_3	Saving Private Ryan
v_4	Schindler's List
v_8	Full Metal Jacket

Type T$_3$ (/film/war)	
e_3	Saving Private Ryan
e_4	Schindler's List
e_8	Heartbreak Ridge

Column C$_4$	
v_5	Shark Night 3D
v_6	Da Vinci Code

Type T$_4$ (/film/thriller)	
e_5	Shark Night
e_6	The Da Vinci Code

Figure 2: Example columns and types.

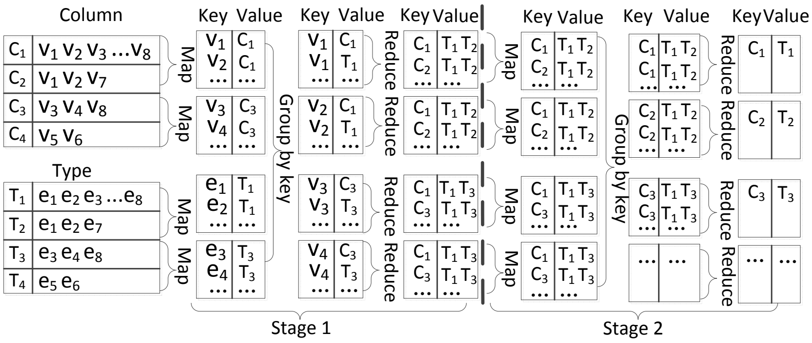


Figure 3: Running example of our framework ($v_1 = e_1$, $v_2 = e_2$, $v_3 = e_3$, $v_4 = e_4$. $v_5 \sim e_5$, $v_6 \sim e_6$ ($\sim$: similar)).

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: $\langle \texttt{C}, \{v_1, \ldots\}\rangle \to \langle \texttt{sig}, \langle \texttt{C}, v_i\rangle\rangle$. $\langle \texttt{T}, \{e_1, \ldots\}\rangle \to \langle \texttt{sig}, \langle \texttt{T}, e_j\rangle\rangle$.

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

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

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 $list(\langle \texttt{T}, v_i, e_j\rangle)$. Finally it outputs key-value pairs $\langle \texttt{C}, list(\langle \texttt{T}, v_i, e_j\rangle)\rangle$.

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

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

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

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

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* C$_4$ *and type* T$_4$. *Suppose the signatures of* v_5 = *"Shark Night 3D" and* e_5 = *"Shark Night" are* sig_1 = *"Shark". The signatures of* v_6 = *"Da Vinci Code" and* e_6 = *"The Da Vinci Code" are* sig_2 = *{"Vinci"}. In the* Map *step of the first stage, we generate pairs* $\langle sig_1, \langle \texttt{C}_4, v_5\rangle\rangle$ *and* $\langle sig_2, \langle \texttt{C}_4, v_6\rangle\rangle$ *for* C$_4$ *and* $\langle sig_1, \langle \texttt{T}_4, v_5\rangle\rangle$ *and* $\langle sig_2, \langle \texttt{T}_4, v_6\rangle\rangle$ *for* T$_4$. *In the* Reduce *step, we generate* $\langle \texttt{C}_4, \langle \texttt{T}_4, v_5, e_5\rangle\rangle$ *and* $\langle \texttt{C}_4, \langle \texttt{T}_4, v_6, e_6\rangle\rangle$. *In the* Reduce *step of the second stage, we compute the fuzzy similarity. The top-1 type of* C$_4$ *is* T$_4$. *Thus the fuzzy matching functions can annotate* 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 $\langle e, \texttt{T}\rangle$ for every type $\texttt{T} \in$

$\mathcal{L}(e)$, we generate key-value pairs $\langle e, \mathcal{G}_i(e)\rangle$ for every group $\mathcal{G}_i(e)$ for $1 \le i \le l$. That is we use group $\mathcal{G}_i(e)$ 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 $\mathcal{L}(e)$ and partition them into l disjoint groups $\mathcal{G}_1(e)$, $\mathcal{G}_2(e)$, …, $\mathcal{G}_l(e)$, such that

(1) Subset: $\mathcal{G}_i(e)$ is a subset of $\mathcal{L}(e)$ for $1 \le i \le l$;

(2) Completeness: $\cup_{1\le i\le l}\mathcal{G}_i(e) = \mathcal{L}(e)$; and

(3) Disjoint: $\mathcal{G}_i(e) \cap \mathcal{G}_j(e) = \phi$ for $i \ne j$.

Let $\mathcal{G}_{\mathcal{T}} = \{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_g\}$ denote the set of distinct groups for all entities. Let $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}$ denote a hash table from the distinct groups to lists of types, i.e., each group $\mathcal{G}_i$ is associated with a list of types contained in $\mathcal{G}_i$, $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}$ $(\mathcal{G}_i)$.

For example, consider the types and entities in Figure 2. $\mathcal{L}(e_1) = \{\texttt{T}_1, \texttt{T}_2\}$. $\mathcal{L}(e_2) = \{\texttt{T}_1, \texttt{T}_2\}$. $\mathcal{L}(e_3) = \{\texttt{T}_1, \texttt{T}_3\}$. $\mathcal{L}(e_4) = \{\texttt{T}_1, \texttt{T}_3\}$. $\mathcal{L}(e_5) = \{\texttt{T}_1, \texttt{T}_4\}$. $\mathcal{L}(e_6) = \{\texttt{T}_1, \texttt{T}_4\}$. $\mathcal{L}(e_7) = \{\texttt{T}_1, \texttt{T}_2\}$. $\mathcal{L}(e_8) = \{\texttt{T}_1, \texttt{T}_3\}$. The basic framework will generate 16 key-value pairs. Suppose we aggregate them into three groups, $\mathcal{G}_1 = \{\texttt{T}_1, \texttt{T}_2\}, \mathcal{G}_2 = \{\texttt{T}_1, \texttt{T}_3\}, \mathcal{G}_3 = \{\texttt{T}_1, \texttt{T}_4\}$. We generate 8 key-value pairs $\langle e_1, \mathcal{G}_1\rangle$, $\langle e_2, \mathcal{G}_1\rangle$, $\langle e_3, \mathcal{G}_2\rangle$, $\langle e_4, \mathcal{G}_2\rangle$, $\langle e_5, \mathcal{G}_3\rangle$, $\langle e_6, \mathcal{G}_3\rangle$, $\langle e_7, \mathcal{G}_1\rangle$, and $\langle e_8, \mathcal{G}_2\rangle$.

Suppose we can get the groups of each entity, i.e., $\langle e, \{\mathcal{G}_1(e)$, $\mathcal{G}_2(e)$,…, $\mathcal{G}_l(e)\}$. 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`: $\langle \texttt{C}, \{v_1, v_2, \ldots\}\rangle \rightarrow \langle v_i, \texttt{C}\rangle$. $\langle e, \{\mathcal{G}_1(e), \ldots, \mathcal{G}_l(e)\}\rangle \rightarrow \langle e, \mathcal{G}_i(e)\rangle$.

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 $\langle e, \{\mathcal{G}_1(e), \mathcal{G}_2(e), \ldots, \mathcal{G}_l(e)\}\rangle$. For each entity, it outputs key-value pairs $\langle e, \mathcal{G}_i(e)\rangle$ for $1 \le i \le l$.

`Reduce`: $\langle v = e, list(\texttt{C}/\mathcal{G})\rangle \rightarrow \langle \texttt{C}, list(\mathcal{G})\rangle$.

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

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

`Map`: $\langle \texttt{C}, list(\mathcal{G})\rangle \rightarrow \langle \texttt{C}, list(\mathcal{G})\rangle$.

Similarly it outputs key-value pairs $\langle \texttt{C}, list(\mathcal{G})\rangle$.

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

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 $\texttt{T} \in \mathcal{G} \in list(list(\mathcal{G}))$. We first initialize a hash table. Then for each group $\mathcal{G}$ in $list(list(\mathcal{G}))$, we get the list of types in the group, i.e., $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}(\mathcal{G})$. For each type in $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}(\mathcal{G})$, we compute its occurrence number similar as the basic framework. Finally we identify top-k types with the largest overlap similarity and add them into topk(T).

Algorithm 2: Aggregation-based Algorithm

```
// the first stage
Map($\langle \texttt{C}/e, \{v_1, v_2, \ldots\}/\{\mathcal{G}_1(e), \mathcal{G}_2(e), \ldots\}\rangle$)
    foreach column $\langle \texttt{C}, \{v_1, v_2, \ldots\}\rangle$ do output($\langle v_i, \texttt{C}\rangle$);
    foreach entity $\langle e, \{\mathcal{G}_1(e), \mathcal{G}_2(e), \ldots\}\rangle$ do
        output($\langle e, \mathcal{G}_i(e)\rangle$);
Reduce ($\langle v = e, list(\texttt{C}/\mathcal{G})\rangle$)
    foreach C/T in $list(\texttt{C}/\mathcal{G})$ do
        if C/T is a group then add $\mathcal{G}$ to $list(\mathcal{G})$;
        else if C/T is a column then add C to $list(\texttt{C})$;
    foreach C in $list(\texttt{C})$ do output($\langle \texttt{C}, list(\mathcal{G})\rangle$);
// the second stage
Map ($\langle \texttt{C}, list(\mathcal{G})\rangle$)
    output($\langle \texttt{C}, list(\mathcal{G})\rangle$);
ReduceSetup ()
    Each reducer loads side data $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}$ from DFSx;
Reduce ($\langle \texttt{C}, list(list(\mathcal{G}))\rangle$)
    Initialize a hash map $\mathcal{H}$;
    foreach $\mathcal{G}$ in $list(list(\mathcal{G}))$ do
        foreach T in $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}(\mathcal{G})$ do
            if $\texttt{T} \in \mathcal{H}$ then $\mathcal{H}(\texttt{T}) = \mathcal{H}(\texttt{T}) + 1$;
            else $\mathcal{H}(\texttt{T}) = 1$;
    topk(T) = top-$k$ types with largest $\mathcal{H}(\texttt{T})$;
    output($\langle \texttt{C}, \texttt{topk(T)}\rangle$);
```

`ReduceSetup`**:** To get $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}$ in each reducer, we take $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}$ as the side data which is distributed to each node of the cluster.

The aggregation based method can correctly identify the top-k types for each column as formalized in Theorem 1.

THEOREM 1. *The aggregation based method satisfies correctness – it can correctly find top-k types for each column.*

PROOF SKETCH. The aggregation based method aggregates the types that contain an entity e (i.e., $\mathcal{L}(e)$) into several disjoint groups $\mathcal{G}_1(e)$, $\mathcal{G}_2(e)$, …, $\mathcal{G}_l(e)$. For each type T and column C, the basic method counts their overlap based on $\langle e \in \texttt{T}, \texttt{T}\rangle$ and $\langle c \in \texttt{C}, \texttt{C}\rangle$. The aggregation method counts their overlap based on $\langle e, \mathcal{G}_i(e)\rangle$ and $\langle c, \texttt{C}\rangle$. As $\cup_{1\le i\le l}\mathcal{G}_i(e) = \mathcal{L}(e)$, and $\mathcal{G}_i(e) \cap \mathcal{G}_j(e) = \phi$ for $i \ne j$, the two counted overlap numbers are the same. Thus the aggregation based method satisfies correctness. □

EXAMPLE 3. *Consider three groups $\mathcal{G}_1 = \{\texttt{T}_1, \texttt{T}_2\}, \mathcal{G}_2 = \{\texttt{T}_1, \texttt{T}_3\}, \mathcal{G}_3 = \{\texttt{T}_1, \texttt{T}_4\}$. Figure 4 shows how to use the aggregation based method to label each column. In the* `Map` *step of the first stage, for each entity, we generate ⟨entity, group⟩ pairs. For e_1, we generate one pair $\langle e_1, \mathcal{G}_1\rangle$. The basic framework generates 16 key-value pairs for the knowledge base while the aggregation based method only generates 8 pairs. In the* `Reduce` *step, we find the groups of types that share common entities with columns. For $\texttt{C}_1$, we get $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$. In the* `Reduce` *step of the second stage, we utilize the side data $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}$ to compute overlap between each type and column. For example, consider $\langle \texttt{C}_1, \{\{\mathcal{G}_1\}, \{\mathcal{G}_1\}, \{\mathcal{G}_2\}, \{\mathcal{G}_2\}\}\rangle$. Based on $\mathcal{G}_1 = \{\texttt{T}_1, \texttt{T}_2\}, \mathcal{G}_2 = \{\texttt{T}_1, \texttt{T}_3\}$, we have the overlap of $\texttt{T}_1$ with $\texttt{C}_1$ is 4, and the overlap of $\texttt{T}_2$ ($\texttt{T}_3$) with column $\texttt{C}_1$ is 2 (2). Thus the top-1 type of column $\texttt{C}_1$ is $\texttt{T}_1$.*

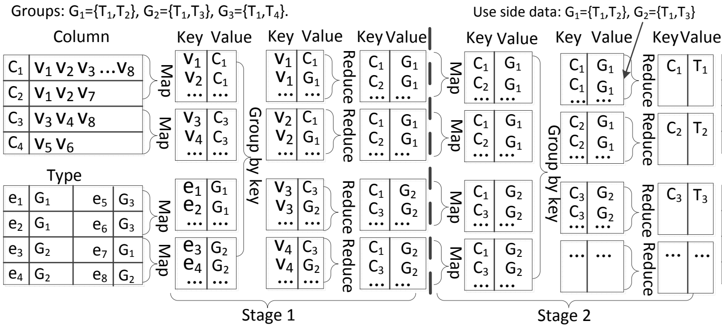

Groups: G1={T1,T2}, G2={T1,T3}, G3={T1,T4}.
Use side data: G1={T1,T2}, G2={T1,T3}
Column
Type
Key Value
Map
Group by key
Reduce
Stage 1
Stage 2

Figure 4: Knowledge type aggregation based method.

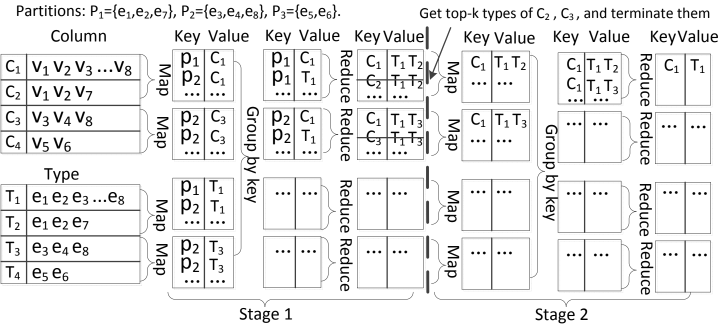

Partitions: P1={e1,e2,e7}, P2={e3,e4,e8}, P3={e5,e6}.
Get top-k types of C2, C3, and terminate them
Column
Type
Key Value
Map
Group by key
Reduce
Stage 1
Stage 2

Figure 5: Knowledge entity partition based method.

Discussion: The aggregation method can reduce the number of key-value pairs for knowledge bases at the expense of distributing the side data. Notice that distributing side data is more efficient than sending key-value pairs. This is because MapReduce has large overhead to generate the pairs, e.g., communication between job tracker and task tracker.

The aggregation method includes two parts: (1) using the groups to generate the key-value pairs for each entity; and (2) distributing the groups $\mathcal{H}_{\mathcal{T}}^{\mathcal{G}}$ to each node of the cluster. Our optimal objective is to select a good aggregation strategy to achieve the highest performance by trading-off them. Here we show two extreme aggregation strategies.

Case 1: For entity e, each type in $\mathcal{L}(e)$ is taken as a group. This strategy still produces large numbers of key-value pairs. However it does not need to distribute the side data.

Case 2: For entity e, the whole list $\mathcal{L}(e)$ is taken as a group. This method produces only one key-value pair for the entity. Thus it can minimize the number of key-value pairs. However it generates large numbers of distinct groups and the side data is very large.

The two strategies cannot balance the number of key-value pairs and the size of side data. To address this issue, we propose a cost model to quantify an aggregation strategy and utilize the cost model to select high-quality groups in the following sections.

5.2 Aggregation Cost Model

The basic framework generates $\langle e, \mathtt{T}\rangle$ key-value pairs. The aggregation based method reduces the number of key-value pairs by aggregating $\langle e, \mathtt{T}\rangle$ pairs on types as follows. For each entity e, it generates a set of distinct groups, $\mathcal{G}_1(e)$, $\mathcal{G}_2(e)$, …, $\mathcal{G}_l(e)$. Let $\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_{|\mathcal{G}|}$ denote the union of groups of all entities. Let $\mathtt{E}_i$ denote the set of entities whose groups include $\mathcal{G}_i$. That is $\mathtt{E}_i = \{e | \exists\ j, \mathcal{G}_i = \mathcal{G}_j(e)\}$. The aggregation based method distributes groups $\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_{|\mathcal{G}|}$ as the side data. For each entity $e \in \mathtt{E}_i$, it generates key-value pairs $\langle e, \mathcal{G}_i\rangle$. Suppose the cost of processing a key-value pair in MapReduce is α and the cost of distributing a group $\mathcal{G}_i$ in the side data is $\beta|\mathcal{G}_i|$ where $|\mathcal{G}_i|$ is the number of types in $\mathcal{G}_i$. Thus the total cost of the aggregation based method is

$$\sum_i \big(\alpha|\mathtt{E}_i| + \beta|\mathcal{G}_i|\big), \tag{1}$$

and the total cost of the basic framework is

$$\sum_i \big(\alpha|\mathtt{E}_i| \times |\mathcal{G}_i|\big). \tag{2}$$

To select the optimal aggregation strategy for achieving the highest performance, we need to minimize Equation 1. It is unlike to find the optimal aggregation strategy in polynomial time, as shown in Theorem 2.

Theorem 2. *The problem of finding the optimal aggregation strategy is NP-hard.*

Proof. We prove the Theorem by a reduction from the 3-CNF-SAT problem. See Appendix for more details. □

Given Side-Data Budget: Usually each node has a memory budget $\mathcal{B}$ for the side data. Our goal is to fully utilize the budge $\mathcal{B}$ to accommodate aggregated groups so as to maximize the performance.

5.3 Aggregation Algorithms

Since the problem of finding the optimal aggregation strategy is NP-hard, we need to devise approximation algorithms. Consider any group $\mathcal{G}_i$. Based on Equations 1 and 2, the aggregation based method can reduce the cost by

$$\alpha|\mathtt{E}_i| \times |\mathcal{G}_i| - (\alpha|\mathtt{E}_i| + \beta|\mathcal{G}_i|).$$

Based on this observation, we devise a greedy algorithm. For each entity e, we enumerate all possible partition of $\mathcal{L}(e)$ into disjoint groups. (Since the number of types that contain an entity is small, the number of possible partitions is not large.) We union all possible groups for all entities and generate a possible group set. Then we enumerate each possible group $\mathcal{G}_i$ in the set and select the group with the largest benefit $\alpha|\mathtt{E}_i| \times |\mathcal{G}_i| - (\alpha|\mathtt{E}_i| + \beta|\mathcal{G}_i|)$. Then we remove the group and repeat the above step until the selected groups cover all entity-type pairs. If given a space budget $\mathcal{B}$, we terminate generating groups if the size of aggregated groups is larger than $\mathcal{B}$.

However there may be large numbers of possible groups, and it is expensive to enumerate all of them. To alleviate this problem, we utilize the knowledge structure to generate the groups as follows. We first pick a type $\mathtt{T}$ with no subtype. Let $\mathcal{G}_i$ denote the set of types that have a directed path to $\mathtt{T}$ (including itself). Obviously any type in $\mathcal{G}_i$ will take $\mathtt{T}$ as its subtype or has a descendant which takes $\mathtt{T}$ as a subtype. Let $\mathtt{E}$ denote the entity set of $\mathtt{T}$. We take $\mathcal{G}_i$ as a group. Then we remove the type $\mathtt{T}$ from the knowledge base and remove the entities in $\mathtt{E}$ from the entity sets of types in $\mathcal{G}_i$. We repeat the above steps until there is no type left (or the size of aggregated types is larger than the budget). Iteratively we can generate all groups. Notice that we need to build key-value pairs $\langle e, \{\mathcal{G}_1(e), \mathcal{G}_2(e), \ldots, \mathcal{G}_l(e)\}\rangle$ for the MapReduce input file and pairs $\langle \mathcal{G}_i, \mathcal{H}_{\mathcal{T}}^{\mathcal{G}}(\mathcal{G}_i)\rangle$ for the side data. We only need to slightly change our algorithm. For each pair $\langle \mathtt{E}, \mathcal{G}_i\rangle$, we generate pair $\langle \mathcal{G}_i, \mathcal{H}_{\mathcal{T}}^{\mathcal{G}}(\mathcal{G}_i)\rangle$. For each entity e in $\mathtt{E}$, if there exists a pair with key e, we add $\mathcal{G}_i$ into its value; otherwise we generate a new pair $\langle e, \mathcal{G}_i\rangle$. Iteratively we can construct the two files (MapReduce input and side data).

For example, consider the four types in Figure 2. Types $\mathtt{T}_2, \mathtt{T}_3, \mathtt{T}_4$ are subtypes of type $\mathtt{T}_1$. Thus we can aggregate $\mathtt{T}_2$ with $\mathtt{T}_1$ and generate a group $\{\mathtt{T}_1, \mathtt{T}_2\}$ which will be inserted into the side data $\mathcal{H}^{\mathcal{G}}_{\mathcal{T}}$. As $\mathtt{T}_2$ contains $\{e_1, e_2, e_7\}$, entities e_1, e_2, e_7 will use the group $\{\mathtt{T}_1, \mathtt{T}_2\}$ and we will add the group into the MapReduce file.

Notice that we can utilize type hierarchies to improve performance and scalability. We can extract hierarchies from existing knowledge bases. For example, we can use "*included types*" [‡] to extract type hierarchy from Freebase. We can also identify the subtype relationships by comparing the entity sets of different types. (For example, if the entity set of a type contains that of another type, the latter type will be a subtype of the former one.) Notice that our techniques do not dependent on type hierarchies and our methods still work if there is no type hierarchy.

6. KNOWLEDGE ENTITY PARTITION

If we can send similar ⟨column, type⟩ pairs into the same reducer and dissimilar ⟨column, type⟩ pairs into different reducers, we can significantly improve the performance since we can prune large numbers of dissimilar ⟨column, type⟩ pairs. To this end, we propose a partition based method. We partition cell values of all columns into p partitions $\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_p$. If all the cell values in a column fall in the same partition, e.g., $\mathtt{P}_i$, we only use this partition to annotate the column and prune other unnecessary partitions. This method not only prunes many unnecessary partitions, but also reduces the number of key-value pairs since $\mathtt{P}_i$ may contain multiple cell values.

There are several research challenges. The first one is how to utilize the partitions to annotate each column. We will discuss a partition framework in Section 6.1. The second challenge is how to effectively partition the cell values. We study how to select a good partition strategy in Section 6.2. The third challenge is how to efficiently identify the corresponding partitions for each column and type, and we propose a bloom-filter based method in Section 6.3.

6.1 Partition Framework

Let $\mathtt{V}$ denote the set of distinct cell values in all columns. We partition $\mathtt{V}$ into p partitions $\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_p$ such that (1) $\mathtt{P}_i$ is a subset of $\mathtt{V}$, (2)$\cup_{1 \le i \le p} \mathtt{P}_i = \mathtt{V}$, and (3) $\mathtt{P}_i \cap \mathtt{P}_j = \phi$. We discuss how to do effective partition in Section 6.2. Here we focus on using the partitions to label each column. We still employ a two stage MapReduce framework and Algorithm 3 shows the pseudo-code. We take the basic framework as an example and our techniques can be easily integrated into the aggregation based method.

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

Map: $\langle \mathtt{C}, \{v_1, v_2, \ldots\} \rangle \rightarrow \langle \mathtt{P}_i, [\mathtt{C}, \mathtt{P}_i \cap \mathtt{V}_\mathtt{C}, \mathtt{cnt} = |\mathtt{V}_\mathtt{C}| - |\mathtt{P}_i \cap \mathtt{V}_\mathtt{C}] \rangle$.
$\langle \mathtt{T}, \{e_1, e_2, \ldots\} \rangle \rightarrow \langle \mathtt{P}_i, [\mathtt{T}, \mathtt{P}_i \cap \mathtt{E}_\mathtt{T}] \rangle$.

For each column $\mathtt{C}$, it identifies the partitions that contain cell values in the column. Since there are large numbers of cell values that cannot be loaded into memory, it is not straightforward to identify the partitions and we will discuss efficient techniques in Section 6.3. Suppose we get k partitions $\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_k$. Notice that if a cell value in $\mathtt{P}_i$ is not an entity in the knowledge base, we do not need to distribute the cell value. In other words, we only distribute the cell values in $\mathtt{C}$ that also appear in knowledge types. For simplicity, we use $\mathtt{V}^\mathtt{T}_\mathtt{C}$ to denote the set of such cell values. Thus we generate the following key-value pairs

Algorithm 3: Partition-based Algorithm

```
// the first stage
Map($\langle$C/T$, \{v_1, v_2, \ldots\}/\{e_1, e_2, \ldots\}\rangle$)
    foreach $\langle$C$, \{v_1, v_2, \ldots\}\rangle$ do
        $\{$P$_1,$P$_2, \ldots,$P$_k\}$ = PartitionIdentification(C);
        output($\langle$P$_i$, C$\rangle$);
    foreach $\langle$T : $\{e_1, e_2, \ldots\}\rangle$ do
        $\{$P$_1,$P$_2, \ldots,$P$_j\}$ = PartitionIdentification(T);
        output($\langle$P$_i$, T$\rangle$);
Reduce ($\langle$P$_i$, $list$(C/T)$\rangle$)
    foreach C/T in $list$(C/T) do
        if C/T is a type then add T to $list$(T);
        else if C/T is a column then add C to $list$(C);
    foreach C in $list$(C) do
        $\tau$=$k$ largest overlap of types in $list$(T) with C;
        $list$(T)=types with overlap $\ge \tau-$cnt;
        if cnt $= 0$ then return $list$(T);
        else output($\langle$C, $list$(T)$\rangle$);
// the second stage is the same as that of the
    basic framework
```

$$\langle \mathtt{P}_i, [\mathtt{C}, \mathtt{P}_i \cap \mathtt{V}^\mathtt{T}_\mathtt{C}, \mathtt{cnt} = |\mathtt{V}^\mathtt{T}_\mathtt{C}| - |\mathtt{P}_i \cap \mathtt{V}^\mathtt{T}_\mathtt{C}] \rangle$$

for $1 \le i \le k$, where $\mathtt{cnt}$ is the number of entities that are in the column but not in the partition. Obviously if $\mathtt{cnt} = 0$, all the cell values appear in partition $\mathtt{P}_i$. In the Reduce step, we can use this partition to directly label the column, because all of types that share entities with the column must fall in this partition.

Similarly, for each type $\mathtt{T}$, it identifies the partitions that contain entities in the type. Suppose we get j partitions $\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_j$. We generate the following key-value pairs

$$\langle \mathtt{P}_i, [\mathtt{T}, \mathtt{P}_i \cap \mathtt{E}_\mathtt{T}] \rangle$$

for $1 \le i \le j$, where $\mathtt{E}_\mathtt{T}$ is the entity set of type $\mathtt{T}$.

Reduce: $\langle \mathtt{P}_i, list(\mathtt{C/T}) \rangle \rightarrow \langle \mathtt{C}, list(\mathtt{T}) \rangle$.

For each partition, we get a list of columns and types. We first extract the list of columns and the list of types. Then for each column $\mathtt{C}$, we scan the type list and for each type $\mathtt{T}$, we compute $(\mathtt{P}_i \cap \mathtt{V}^\mathtt{T}_\mathtt{C}) \cap (\mathtt{P}_i \cap \mathtt{E}_\mathtt{T})$, which is the overlap similarity of $\mathtt{C}$ and $\mathtt{T}$ in the partition. Let τ denote the k-th largest overlap similarly among all types with column $\mathtt{C}$. Since column $\mathtt{C}$ contains at most other $\mathtt{cnt}$ entities besides the partition, each type $\mathtt{T}$ at most includes other $\mathtt{cnt}$ cell values of $\mathtt{C}$. Thus we can prune the types with the overlap smaller than $\tau - \mathtt{cnt}$. In other words, we generate a list of types, $list(\mathtt{T})$, where each type has no smaller than $\tau - \mathtt{cnt}$ overlap with the column, and output key-value pairs $\langle \mathtt{C}, list(\mathtt{T}) \rangle$. Especially, if $\mathtt{cnt} = 0$, column $\mathtt{C}$ contains no other cell value and all types having overlap with $\mathtt{C}$ are also in this partition, thus we can directly use the partition in this reducer to label the column and do not need to output the pair.

Stage 2: Compute top-k types of every column. It is the same as that of the basic framework.

EXAMPLE 4. *Suppose we partition the entities into three*

[‡] http://wiki.freebase.com/wiki/Included_Type

partitions $\mathtt{P}_1 = \{e_1, e_2, e_7\}, \mathtt{P}_2 = \{e_3, e_4, e_8\}, \mathtt{P}_3 = \{e_5, e_6\}$. *Figure 5 illustrates a running example of using the partition-based method to label the columns. In the* Map *step of the first stage, it generates* ⟨*partition, entity*⟩ *and* ⟨*partition, column*⟩ *pairs. For* $\mathtt{T}_1$*, it generates three pairs* $\langle \mathtt{P}_1, \mathtt{T}_1\rangle$, $\langle \mathtt{P}_2, \mathtt{T}_1\rangle$, $\langle \mathtt{P}_3, \mathtt{T}_1\rangle$. *The basic framework generates 16 pairs for the knowledge base while the partition based method only generates 6 pairs. For column* $\mathtt{C}_2$*, it generates one pair* $\langle \mathtt{P}_1, \mathtt{C}_2\rangle$. *As all cell values fall in partition* $\mathtt{P}_1$*, we can use this partition to label the column in the* Reduce *step. Similarly all cell values in* $\mathtt{C}_3$ *fall in partition* $\mathtt{P}_2$. *The basic framework generates 16 pairs for the columns while the partition method only generates 4 pairs* (v_5, v_6, v_7, v_8 *do not appear in the knowledge base). In the* Reduce *step, in the reducer of* $\mathtt{P}_1$*, we can directly label* $\mathtt{C}_2$ *and do not need to output the key-value pairs since all types having overlap with* $\mathtt{C}_2$ *are also in this partition. Similarly we can directly label* $\mathtt{C}_3$ *and early terminate the two columns.*

Notice that the partition based method is orthogonal to the aggregation based method and the two methods can be used simultaneously.

6.2 Partition Algorithm

Since different partition strategies will affect the pruning power (whether we can prune unnecessary partitions) and the number of key-value pairs, it is very important to select a high-quality partition strategy. To address this issue, we first study how to evaluate a partition strategy.

To evaluate a partition strategy, we model the columns as a graph and transform the graph partition problem to our partition problem as follows. For ease of presentation, we first assume each column has at most two cell values. In the graph, the vertices are cell values. There is an edge between two vertices if there is a column which contains the cell values of the two vertices. The edge weight is the number of columns that contain the two cell values. If we want to partition the cell values into p partitions, we aim to partition the graph into p subgraphs. Notice that in a graph partition strategy, if we cut an edge with weight w, w columns w.r.t. the edge will be affected. Thus cutting an edge with weight w will increase w key-value pairs. It also reduces the probability to use a single partition to label a column. Thus we want to find a partition with minimum cut weight, i.e., the sum of the weights of cut edges. Thus we can transform the problem of graph partition problem to our finding optimal partition strategy problem.

We can extend this method to support the case that each column can have more than two cell values. The only difference is that the weight of cutting an edge is dependent on those edges that have been cut. We can also utilize the graph partition method to evaluate a partition strategy. However note that the Web tables are not given and we cannot partition the cell values off-line. To address this problem, we use the entities in the knowledge base to facilitate the partition. The main reason is that if a cell value does not appear in the knowledge base, we cannot utilize it to do annotation. Thus we can utilize the entities to construct the graph. In addition, given an edge of two entities, we cannot efficiently get its weight, i.e., the real number of columns that contain the two entities. Alternatively, we can use the number of types that contain the two entities to approximate the edge weight. Thus we can build a graph based on the knowledge base. We extend existing graph-partitioning algorithms (e.g., [16]) to partition the weighted graph, with the goal of (1) minimizing the weight of cut edges and (2) making each partition nearly have the same weight.

Figure 6 shows the graph constructed from the knowledge base in Figure 2. Each vertex represents a distinct entity. The weight of an edge is the number of types that contain the two entities on the edge. The weight of edges between two dotted ellipses denotes the weights of all the edges between vertices in the two ellipses. If $p = 3$, the three partitions should be $\mathtt{P}_1 = \{e_1, e_2, e_7\}, \mathtt{P}_2 = \{e_3, e_4, e_8\}, \mathtt{P}_3 = \{e_5, e_6\}$.

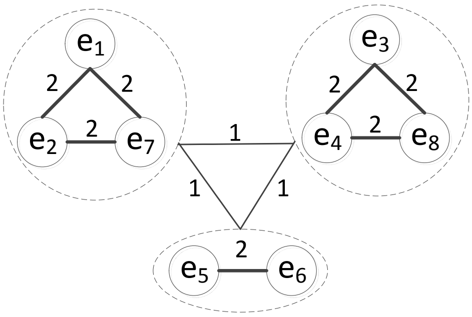


Figure 6: Partitioning entities.

6.3 Partition Identification

In this section, we discuss how to identify partitions for each type and column. For each type, during the graph partitioning phase, we keep the partitions for each type and generate the key-value pair $\langle \mathtt{T}, \{\mathtt{P}_1, \mathtt{P}_2, \ldots\}\rangle$. Thus in the Map step of the first stage, we directly load the pairs to generate the key-value pairs $\langle \mathtt{P}_i, \mathtt{T}\rangle$.

Since each column is not given, we need to online identify the partitions that contain a cell value in the column. A straightforward method is to use a hash based method. However the number of entities is rather large, and we cannot load the hash table into memory. To address this issue, we use a bloom filter based method. The bloom filter [4] is a probabilistic method to efficiently check whether an element is in a set using multiple hash functions.

For each partition, we use a bloom filter to represent the entities in the partition. To identify the partitions of each column, for each cell value, we enumerate the p partitions. For each partition, we check whether the cell value is in the partition. If the cell value is not in the partition based on the bloom filter, we can ignore the partition; otherwise the partition is added into the partition list of the column, with a false-positive probability. Iteratively we can identify all the partitions. The false positives introduced by the bloom filter will not affect the correctness as stated in Theorem 3.

THEOREM 3. *The partition based method satisfies correctness – it can correctly find top-k types for each column.*

PROOF SKETCH. The partition based method partitions the types into $\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_p$. If a cell value is the same as an entity, they must be in a same partition. Thus if a type T and a column C share a common entity/cell value, they must share a common partition. Although the bloom filter may involve false positives, we compute their overlap based on $(\mathtt{P}_i \cap \mathtt{V}^{\mathtt{T}}_{\mathtt{C}}) \cap (\mathtt{P}_i \cap \mathtt{E}_{\mathtt{T}})$. If a cell value is a false positive of partition $\mathtt{P}_i$, it will not be in $\mathtt{P}_i \cap \mathtt{E}_{\mathtt{T}}$. Thus the partition based method satisfies correctness. □

However this method needs to enumerate p bloom filters of every partition and for each partition, it has to compute

t hash values using the t hash functions, which is expensive. To alleviate this problem, we utilize a bloom filter hierarchy [24] to improve the performance. We build a binary tree structure where each leaf node is a bloom filter. The parent of two leaf nodes is the bloom filter build upon entities in the two partitions. Obviously the root is the bloom filter for all entities. Then given a cell value, we identify the partitions in an up-down manner from the root. If the cell value is not in the root bloom filter, we terminate; otherwise we check its two children. Iteratively, we can identify all partitions. This method only needs to enumerate $\log_2 p$ bloom filters.

7. EXPERIMENTS

The goals of our experiments are to (1) evaluate the effectiveness of our column concept identification approach that utilizes knowledge bases fuzzily to label Web tables; and (2) validate the performance and scalability of our methods.

Datasets: Table 2 summarizes our datasets. For Web tables, we used (1) `WWT`[§], whose tables were manually extracted from Wikipedia and annotated with ground-true entities and types, and (2) `WEX`[¶], whose tables were extracted from Wikipedia by a Freebase Wikipedia Extraction project. For knowledge bases, we used (1) `Freebase`[‖], a collaboratively created knowledge based that is manually built by data-loving users, and (2) `Yago`[**], a large semantic knowledge base which is derived from Wikipedia, WordNet and GeoNames. Both knowledge bases are publicly well-known.

Table 2: Web Tables and Knowledge Bases

Dataset	Size (MB)	# Tables	# Columns	# Cell Values
`WWT`	7.3	6318	27,695	459,641
`WEX`	423	242,062	1,179,125	19,922,138

Dataset	Size (MB)	# Types	# Entities
`Freebase`	1200	1721	43,648,334
`Yago`	157	292,861	9,003,751

Settings: We implemented our algorithms in Hadoop. All experiments except scaleup ones were run on a 20-node Dell cluster, each with two Intel(R) Xeon(R) E5420 2.5GHZ processors with eight cores, 16GB RAM, and 1TB disk. Thus the cluster consists of 160 cores, 320GB memory and 20T-B disk. Each node is installed with 64-bit Ubuntu Server 10.04, Java 1.6, and Hadoop 1.0.3. We set $\mathcal{B}$ as 1GB. α and β can be set by evaluating the cost of distributing key-value pairs and side data based on experimental evaluation. In our experiments, we set β as 1 and α as the ratio of the number of entities to the number of distinct entities.

Due to space constraints, in the experiments we only showed the results on the overlap similarity, Jaccard similarity, Fuzzy Jaccard similarity. We obtained similar results for other similarity functions, e.g., Cosine and Dice.

7.1 Annotation Quality

In this section, we evaluate the annotation quality of our similarity-join based framework. We compared with the machine learning based method [18], `GraphicalModel`. We used the well-known metrics, precision, recall, and F-measure, to evaluate the annotation quality. Let $\mathtt{c}_{\mathtt{no}}$ denote the number of columns that have been manually annotated with ground truth, $\mathtt{a}_{\mathtt{no}}$ denote the number of annotated columns of an algorithm, and $\mathtt{a}_{\mathtt{t}}$ denote the number of truly annotated columns (If the ground truth of a column is in the top-k types annotated by an algorithm, the column is taken as truly annotated.). The precision is $p = \frac{\mathtt{a}_{\mathtt{t}}}{\mathtt{a}_{\mathtt{no}}}$, the recall is $r = \frac{\mathtt{a}_{\mathtt{t}}}{\mathtt{c}_{\mathtt{no}}}$, and the F-measure is $\frac{2 \cdot p \cdot r}{p+r}$.

We first compared the quantity by varying k on the `WWT` and `Freebase` datasets. Figure 7 shows the result quality. We have the following observations. First, our similarity join based method, especially using Jaccard similarity and fuzzy Jaccard similarity, nearly achieved the same quality as the machine learning based method `GraphicalModel`. Second, for precision, our method using fuzzy Jaccard similarity outperformed `GraphicalModel`. This is because for each column, `GraphicalModel` will identify top-k types which may be wrongly annotated while our method will not annotate the columns if it cannot find relevant types. In terms of recall, `GraphicalModel` outperformed our similarity join based method since it can always identify top-k types for each column. Third, our method using fuzzy Jaccard similarity was better than that using Jaccard similarity, which in turn outperformed that using the overlap similarity. The is because fuzzy Jaccard similarity can tolerate inconsistences between cell values and entities, and Jaccard similarity utilized the type and column sizes to annotate a column which can remove the *stop* types, e.g., `music`. For example, fuzzy Jaccard similarity outperformed Jaccard similarity about 5%, which was better than overlap similarity about 5%. Fourth, a larger k will increase the annotation quality, because it returned more types to annotate a column.

We then evaluated the quality on different datasets by fixing $k = 3$. Figure 8 shows the results. We have the following observations. First, on different Web tables and knowledge bases, our similarity join based method still achieved the same high quality as `GraphicalModel`. Second, the annotation quality using `Yago` is better than that using `Freebase`, because `Yago` contains larger numbers of types than `Freebase`.

7.2 Performance and Scalability

To evaluate the performance of our algorithms on large datasets, we increased `WEX` by 5, 10, 15, 20, 25 times as follows. Suppose we want to increase a dataset by n times. For each column with m cell values, we randomly selected $m-1, m-2, \ldots, m-n$ cell values and took them as new columns. (If a column had smaller than n cell values, we randomly selected n groups of $m-1$ cell values and took them as new columns.) We use `WEX` $\times n$ to represent the increase factor. For example, `WEX` $\times 10$ represents the `WEX` dataset increased 10 times. Notice that `WEX` $\times 25$ contains about 30 million columns and 500 million cell values.

7.2.1 Running Time

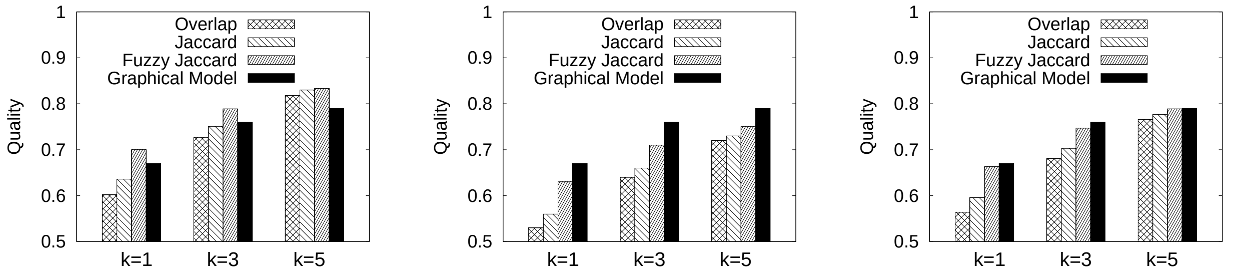
We compare the annotation performance of our four methods: the basic similarity-join framework (`Basic`), the knowledge type aggregation based method (`Aggregation`), the knowledge entity partition based method (`Partition`), and the hybrid method which uses these two optimization techniques (`Hybrid`). Notice that `GraphicalModel` is rather inefficient. For example, it took an hour to annotate 100 tables using `Freebase`. Thus we did not compare with it in terms of running time. We compared the performance by varying the dataset sizes. Since overlap similarity and Jaccard similarity nearly achieved the same performance, we only used

[§] http://www.it.iitb.ac.in/~sunita/wwt/

[¶] http://wiki.freebase.com/wiki/WEX

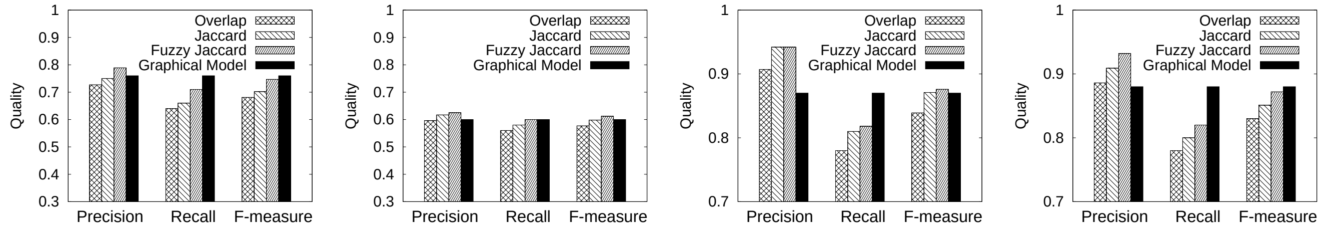
[‖] http://www.freebase.com

[**] http://www.mpi-inf.mpg.de/yago-naga/yago/



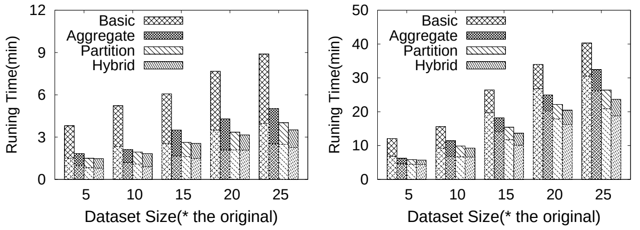
(a) Precision. (b) Recall. (c) F-measure.

Figure 7: Evaluating annotation quality by varying k on `WWT` and `Freebase` datasets.



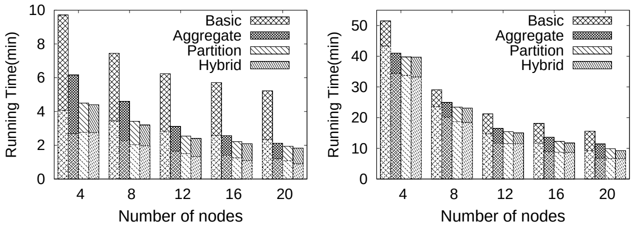
(a) `WWT` + `Freebase` (b) `WEX` + `Freebase` (c) `WWT` + `Yago` (d) `WEX` + `Yago`

Figure 8: Evaluating annotation quality on different datasets ($k = 3$).



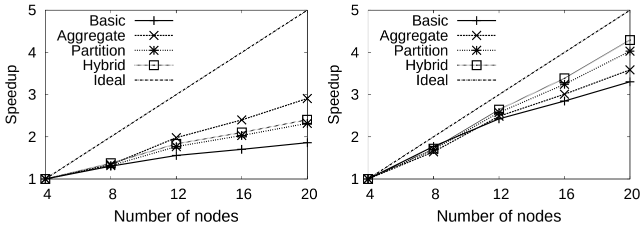
(a) Jaccard (b) Fuzzy Jaccard

Figure 9: Evaluating running time by varying dataset sizes on `WEX` + `Freebase` ($k = 3$).



(a) Jaccard (b) Fuzzy Jaccard

Figure 10: Evaluating running time by varying number of nodes on `WEX` ×10 + `Freebase` ($k = 3$).



(a) Jaccard (b) Fuzzy Jaccard

Figure 11: Speedup on `WEX` ×10 + `Freebase` ($k = 3$).

the Jaccard and fuzzy Jaccard similarity. We used `WEX` and `Freebase` datasets and set $k = 3$. Figure 9 shows the results. In this figure, for each algorithm, the bottom bar denotes the time of the first stage in the algorithm and the top bar denotes the time of the second stage.

We can see that the `Hybrid` method achieved the highest performance and the `Basic` method got the worst performance. The knowledge type aggregation based method improved the performance as it can reduce the number of key-values pairs by aggregating the types. The knowledge entity partition based method also improved the performance, because it cannot only prune large numbers of unnecessary partitions but also reduce the number of key-value pairs. For example, on `WEX` ×25 dataset, `Basic` took about 9 minutes, `Aggregation` improved the time to 5 minutes, and `Partition` took about 4 minutes. `Hybrid` further improved the time to 3.5 minutes. The experimental results show the superiority of our proposed techniques.

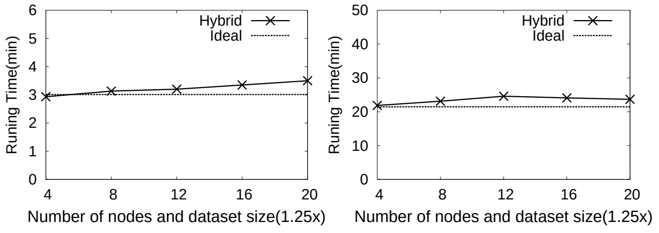
7.2.2 Speedup

We evaluated our algorithms by fixing the datasset size and varying the number of nodes in the cluster. We used `WEX` ×10 and `Freebase` datasets. Figure 10 shows the running time by varying the number of nodes from 4 to 20. We can see that with the increase of the number of nodes, the performance of our method improved. For instance, using Jaccard similarity, `Hybrid` took 4.5 minutes on 4 nodes and improved the time to less than 2 minutes for 20 nodes.

We also evaluated the speedup of our algorithms, i.e., the ratio of the performance improvement compared with that of the minimum number of nodes. For example, for 20 nodes, the speedup is the ratio between the running time on the 4 nodes and that on 20 nodes. We also plot the ideal speedup curve. For example, if the nodes are doubled, the ideal speedup should be 2. Figure 11 shows the results. We can see that our methods achieved good speedup.

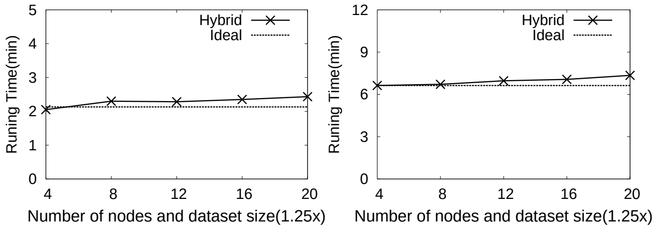
7.2.3 Scaleup

To evaluate the scaleup of our methods, we increased both the dataset sizes and the number of nodes in the cluster. We used our best algorithm, `Hybrid`, which used both knowledge



(a) Jaccard (b) Fuzzy Jaccard

Figure 12: Scaleup by increasing the dataset sizes and the number of nodes on WEX + Freebase ($k = 3$).



(a) Jaccard (b) Fuzzy Jaccard

Figure 13: Scaleup by increasing the dataset sizes and the number of nodes on WEX + Yago ($k = 3$).

type aggregation and entity partition techniques. We also added the ideal scaleup curve which should be a constant. Figures 12 and 13 show the running time on the Freebase and Yago datasets respectively, with increasing the number of nodes from 4 to 20 and dataset sizes from 5 to 25. We can see that our method scaled up very well for both Jaccard similarity and fuzzy Jaccard similarity, and nearly reached the ideal scaleup. This is attributed to our partition based method which can prune unnecessary partitions and reduce the number of key-value pairs.

8. CONCLUSION

We studied the column concept determination problem for Web tables using knowledge bases. We enabled fuzzy matching between the cell content and knowledge entities. We proposed a knowledge type aggregation based method which aggregated the types into groups and shared the groups for different entities to reduce the number of key-value pairs. We proved that the problem of finding the optimal aggregation strategy is NP-hard. We utilized the knowledge hierarchy to effectively aggregate types. We proposed a knowledge entity partition based method which partitioned the entities into different partitions. For each column, we identified its partitions and used the types falling in these partitions to annotate the column. We proposed a graph partition based algorithm to generate high-quality partitions. Experimental results on real datasets show that our algorithm achieved high annotation quality and performance and scaled well.

9. REFERENCES

[1] S. Abiteboul, E. Antoine, and J. Stoyanovich. Viewing the web as a distributed knowledge base. In *ICDE*, 2012.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. In *ISWC/ASWC*, pages 722–735, 2007.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, 2008.

[6] M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *PVLDB*, 1(1):538–549, 2008.

[7] S. Chaudhuri, others Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[9] H. Elmeleegy, J. Madhavan, and A. Y. Halevy. Harvesting relational tables from lists on the web. *PVLDB*, 2(1):1078–1089, 2009.

[10] Google. Introducing knowledge graph. http://insidesearch.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html.

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

[12] X. Guo, Y. Chen, J. Chen, and X. Du. ITEM: Extract and integrate entities from tabular data to RDF knowledge base. In *APWeb*, 2011.

[13] R. Gupta and S. Sarawagi. Answering table augmentation queries from unstructured lists on the web. *PVLDB*, 2(1):289–300, 2009.

[14] L. Han, T. Finin, C. S. Parr, J. Sachs, and A. Joshi. Rdf123: From spreadsheets to rdf. In *ISWC*, 2008.

[15] G. Hignette, P. Buche, J. Dibie-Barthélemy, and O. Haemmerlé. Fuzzy annotation of web data tables driven by a domain ontology. In *ESWC*, 2009.

[16] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *DAC*, 1999.

[17] Y. Kim and K. Shim. Parallel top-k similarity join algorithms using mapreduce. In *ICDE*, 2012.

[18] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 3(1):1338–1347, 2010.

[19] X. Ling, A. Halevy, F. Wu, and C. Yu. Synthesizing union tables from the web. In *IJCAI*, 2013.

[20] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

[21] R. Pimplikar and S. Sarawagi. Answering table queries on the web using column keywords. *PVLDB*, 5(10):908–919, 2012.

[22] G. Quercini and C. Reynaud. Entity discovery and annotation in tables. In *EDBT*, 2013.

[23] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

[24] K. Shanmugasundaram, H. Brönnimann, and N. D. Memon. Payload attribution via hierarchical bloom filters. In *CCS*, 2004.

[25] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[26] M. van Assem, others Converting and annotating quantitative data tables. In *ISWC*, 2010.

[27] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering semantics of tables on the web. *PVLDB*, 4(9):528–538, 2011.

[28] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[29] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, 2011.

[30] J. Wang, H. Wang, Z. Wang, and K. Q. Zhu. Understanding tables on the web. In *ER*, 2012.

[31] W. Wu, others Probase: a probabilistic taxonomy for text understanding. In *SIGMOD*, 2012.

[32] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[33] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.

[34] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*, 2012.

[35] C. Yu. Towards a high quality and web-scalable table search engine. In *KEYS*, 2012.

Appendix

We formally show that the problem of finding the optimal aggregation strategy is NP-hard, even when $\alpha = \beta = 1$. The corresponding decision problem can be defined as below.

Definition 1. *(The decision problem) Given a set of entities $\mathcal{H}$ where each entity associated with a set of types, and a positive threshold t, the decision problem $\langle \mathcal{H}, t \rangle$ returns true if and only if there exists an aggregation strategy that aggregates the set of types of each entity into several groups such that*

$$\sum_{distinct\ group\ G} (|E| + |G|) \leq t$$

where G is a distinct group and E is the set of entities in group G.

For example, in Figure 14(a), let $e_1 = \{T_1, T_2, T_3, T_4, T_5, T_8\}$, $e_2 = \{T_2, T_3, T_6, T_8\}$, $e_3 = \{T_4, T_5, T_7, T_8\}$, and $t = 16$. The decision problem $\langle \mathcal{H} = \{e_1, e_2, e_3\}, t \rangle$ should return yes since the aggregation strategy $e_1 = \{\langle T_2, T_3, T_8 \rangle \langle T_1, T_4, T_5 \rangle\}$, $e_2 = \{\langle T_2, T_3, T_8 \rangle \langle T_6 \rangle\}$ and $e_3 = \{\langle T_4, T_5, T_7, T_8 \rangle\}$ incurs a cost of 16.

We reduce the 3-CNF-SAT problem to the decision problem. A 3-CNF formula ϕ consists of variables $x_1, x_2, \ldots x_n$ and clauses $C_1, C_2, \ldots C_k$. Each clause contains exactly three literals. A literal $l_m (1 \leq m \leq 3)$ is either a variable itself x_i or its negation $\overline{x}_i$. The 3-CNF-SAT problem asks whether there exists an truth assignment of the variables such that the formula ϕ is satisfied. Before proving the decision problem is NP-complete, we first discuss variable gadget $\mathcal{X}^g$ and clause gadget $\mathcal{C}^g$, which will be important in our reduction.

Variable Gadgets: For each variable x in the 3-CNF formula, we construct a variable gadget $\mathcal{X}^g$ which contains three entities $e_1 = \{T_1, T_2, T_3, T_4, T_5, T_8\}$, $e_2 = \{T_2, T_3, T_6, T_8\}$ and $e_3 = \{T_4, T_5, T_7, T_8\}$. T_6 and T_7 are two special nodes we call *tips*. We label tip T_6 with the variable x and tip T_7 with $\overline{x}$. See Figure 14(a) for an illustration. By enumerating all possible aggregation strategies of the variable gadget $\mathcal{X}^g$, we can show that it has the following nice property:

Property 1. *The minimum aggregation cost of variable gadget $\mathcal{X}^g$ is 16 and for each aggregation strategy with cost 16, there must exist one and only one group which contains one tip, T_6 or T_7.*

For a minimum cost aggregation strategy for a particular variable gadget $\mathcal{X}^g$, we let G_x denote the group that contains exactly one tip. The following definitions of $\mathcal{S}_{V_1}$ and $\mathcal{S}_{V_2}$ will be useful in later discussions. Intuitively, $\mathcal{S}_{V_1}$ corresponds to assigning the variable True while $\mathcal{S}_{V_2}$ corresponds to assigning the variable False.

$\mathcal{S}_{V_1}$: We denote as $\mathcal{S}_{V_1}$ the aggregation strategy $e_1 = \{\langle T_2, T_3, T_8 \rangle \langle T_1, T_4, T_5 \rangle\}$, $e_2 = \{\langle T_2, T_3, T_8 \rangle \langle T_6 \rangle\}$, $e_3 = \{\langle T_4, T_5, T_7, T_8 \rangle\}$. The cost is 16 and $G_x = \langle T_6 \rangle$.

$\mathcal{S}_{V_2}$: We denote as $\mathcal{S}_{V_2}$ the aggregation strategy $e_1 = \{\langle T_1, T_2, T_3 \rangle \langle T_4, T_5, T_8 \rangle\}$, $e_2 = \{\langle T_2, T_3, T_8, T_6 \rangle\}$, $e_3 = \{\langle T_4, T_5, T_8 \rangle \langle T_7 \rangle\}$. Its cost is 16 and $G_x = \langle T_7 \rangle$.

Clause Gadgets: For each clause $C = (l_1 \wedge l_2 \wedge l_3)$, we have a clause gadget $\mathcal{C}^g$ which contains seven entities $e_4 = \{T_9, T_{10}, T_{11}\}$, $e_5 = \{T_{11}, T_{13}, T_{14}\}$, $e_6 = \{T_{10}, T_{12}, T_{13}\}$, $e_7 = \{T_{10}, T_{11}, T_{13}\}$, $e_8 = \{T_{10}, T_{l_1}\}$, $e_9 = \{T_{11}, T_{l_2}\}$ and $e_{10} = \{T_{13}, T_{l_3}\}$ where $T_{l_1} \neq T_{l_2} \neq T_{l_3}$. See Figure 14(b). We call T_{l_1}, T_{l_2} and T_{l_3} the *tips* of the clause gadget and they corresponds to literals l_1, l_2 and l_3 in clause C respectively. In the final construction, each tip of the clause gadgets will be identified with the corresponding tip of the variable gadget. This introduces some subtlety in counting the aggregation cost as follows: there may exist a group G_C which contains exactly one tip (T_{l_1}, T_{l_2} or T_{l_3}) in the clause gadget. To avoid double counting the cost of $|G_C| = 1$ in both variable gadget and clause gadget, we do not count it in the clause gadget, but instead treat it as an additional cost c. We let $c = 0$ if (1) G_C exists in both the clause gadget and some variable gadget or (2) No group contain exactly one tip [††]; Otherwise the additional cost $c = 1$. After enumerating all possible aggregation strategies, we observe the following nice property of a clause gadget:

Property 2. *The minimum aggregation cost of clause gadget $\mathcal{C}^g$ is 24 and for each aggregation strategies with cost 24, there must exist a group which only contains one of T_{l_1}, T_{l_2} and T_{l_3}.*

We denote the group in clause gadget $\mathcal{C}^g$ that only contains one of T_{l_1}, T_{l_2} and T_{l_3} as G_C. As before, the following notations will be useful.

$\mathcal{S}_{L_1}$: We denote as $\mathcal{S}_{L_1}$ the strategy $e_4 = \{\langle T_9, T_{11} \rangle \langle T_{10} \rangle\}$, $e_5 = \{\langle T_{11}, T_{13} \rangle \langle T_{14} \rangle\}$, $e_6 = \{\langle T_{10} \rangle \langle T_{12}, T_{13} \rangle\}$, $e_7 = \{\langle T_{11}, T_{13} \rangle \langle T_{10} \rangle\}$, $e_8 = \{\langle T_{10} \rangle \langle T_{l_1} \rangle\}$, $e_9 = \{\langle T_{11}, T_{l_2} \rangle\}$ and $e_{10} = \{\langle T_{13}, T_{l_3} \rangle\}$. Its cost is 24 and $G_C = \langle T_{l_1} \rangle$.

$\mathcal{S}_{L_2}$: We denote as $\mathcal{S}_{L_2}$ the strategy $e_4 = \{\langle T_9, T_{10} \rangle \langle T_{11} \rangle\}$, $e_5 = \{\langle T_{11} \rangle \langle T_{13}, T_{14} \rangle\}$, $e_6 = \{\langle T_{10}, T_{13} \rangle \langle T_{12} \rangle\}$, $e_7 = \{\langle T_{10}, T_{13} \rangle \langle T_{11} \rangle\}$, $e_8 = \{\langle T_{10}, T_{l_1} \rangle\}$, $e_9 = \{\langle T_{11} \rangle \langle T_{l_2} \rangle\}$ and $e_{10} = \{\langle T_{13}, T_{l_3} \rangle\}$. Its cost is 24 and $G_C = \langle T_{l_2} \rangle$.

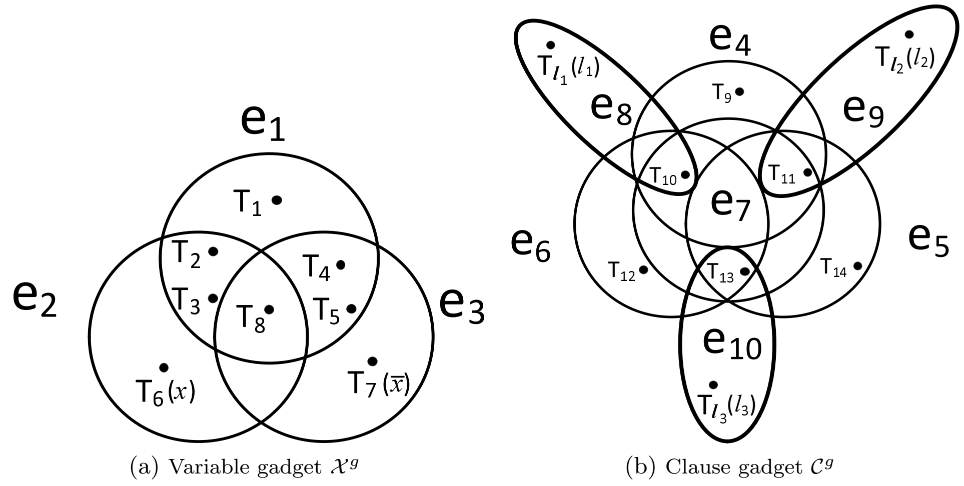
$\mathcal{S}_{L_3}$: We denote as $\mathcal{S}_{L_3}$ the strategy $e_4 = \{\langle T_9 \rangle \langle T_{10}, T_{11} \rangle\}$, $e_5 = \{\langle T_{11}, T_{14} \rangle \langle T_{13} \rangle\}$, $e_6 = \{\langle T_{13} \rangle \langle T_{12}, T_{10} \rangle\}$, $e_7 = \{\langle T_{10}, T_{11} \rangle \langle T_{13} \rangle\}$, $e_8 = \{\langle T_{10}, T_{l_1} \rangle\}$, $e_9 = \{\langle T_{11}, T_{l_2} \rangle\}$ and $e_{10} = \{\langle T_{13} \rangle \langle T_{l_3} \rangle\}$. Its cost is 24 and $G_C = \langle T_{l_3} \rangle$.

We can reduce the well-known NP-complete problem 3-CNF-SAT to our decision problem in polynomial time. For ease of understanding, we first give an example to show the reduction from 3-CNF formula to our decision problem in Figure 15. The formula in 3-CNF is $\phi = C_1 \bigwedge C_2$ where $C_1 = (\overline{x}_1 \bigvee x_2 \bigvee x_4)$ and $C_2 = (\overline{x}_2 \bigvee x_3 \bigvee \overline{x}_4)$. The entity set $\mathcal{H} = \mathcal{X}_1^g \cup \mathcal{X}_2^g \cup \mathcal{X}_3^g \cup \mathcal{X}_4^g \cup \mathcal{C}_1^g \cup \mathcal{C}_2^g$ as show in Figure 15. The set of types associated with each entity is shown in the figure. The variables x_1, x_2, x_3, and x_4 map to the variable gadgets $\mathcal{X}_1^g$, $\mathcal{X}_2^g$, $\mathcal{X}_3^g$, $\mathcal{X}_4^g$ respectively and the clauses C_1 and C_2 map to the clause gadgets $\mathcal{C}_1^g$ and $\mathcal{C}_2^g$ respectively. The literal in the clause gadget is exactly the matching literal in the variable gadget as show in the figure. For example T_7^1 is exactly $T_{l_1}^1$. Here $t = 16n + 24k = 16 * 4 + 24 * 2 = 112$ where n is the number of distinct variable in ϕ and k is the number of clause in ϕ. A satisfied assignment of ϕ is $x_1 = 0$, $x_2 = 0$, $x_3 = 0$ and $x_4 = 0$. The corresponding aggregation strategy is that we aggregate $\mathcal{X}_1^g$, $\mathcal{X}_2^g$, $\mathcal{X}_3^g$ and $\mathcal{X}_4^g$ by $\mathcal{S}_{V_2}$ and aggregate $\mathcal{C}_1^g$ and $\mathcal{C}_2^g$ by $\mathcal{S}_{L_1}$.

Next we formally prove Theorem 4.

Theorem 4. *The decision problem in Definition 1 is NP-Complete.*

[††] In the first case, the cost of $|G_C|$ is counted in the variable gadget. In the second case, there is no such G_C. So the additional cost is also 0.



(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.

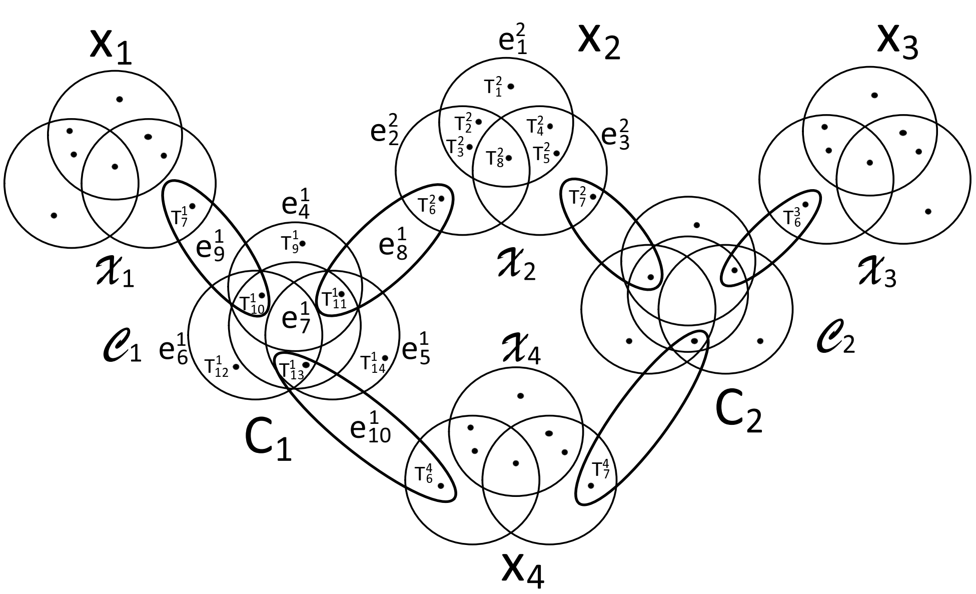


Figure 15: An example of reducing 3-CNF to finding aggregation strategy problem.

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 $1 \le i \le n$, 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 $\mathcal{X}_i^g (1 \le i \le n)$, the total cost is $16n$. For each $1 \le j \le k$, as ϕ is satisfied, C_j contains at least one literal $l_m (1 \le m \le 3)$ 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 $\mathcal{X}_i^g (1 \le i \le n)$, the minimum aggregation cost is 16. Thus the minimum cost to aggregate all variable gadgets is $16n$. For any clause gadget $\mathcal{C}_j^g (1 \le j \le k)$, 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 $c_j (1 \le j \le k)$ 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 $C_j (1 \le j \le k)$ 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. □