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 $1 \leq i \leq l$;

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

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

Let $\mathcal{G}_T = \{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_g\}$ denote the set of distinct groups for all entities. Let $\mathcal{H}_T^{\mathcal{G}}$ 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 $\mathcal{G}_i$, $\mathcal{H}_T^{\mathcal{G}}(\mathcal{G}_i)$.

For example, consider the types and entities in Figure 2. $\mathcal{L}(e_1) = \{\mathsf{T}_1, \mathsf{T}_2\}$. $\mathcal{L}(e_2) = \{\mathsf{T}_1, \mathsf{T}_2\}$. $\mathcal{L}(e_3) = \{\mathsf{T}_1, \mathsf{T}_3\}$. $\mathcal{L}(e_4) = \{\mathsf{T}_1, \mathsf{T}_3\}$. $\mathcal{L}(e_5) = \{\mathsf{T}_1, \mathsf{T}_4\}$. $\mathcal{L}(e_6) = \{\mathsf{T}_1, \mathsf{T}_4\}$. $\mathcal{L}(e_7) = \{\mathsf{T}_1, \mathsf{T}_2\}$. $\mathcal{L}(e_8) = \{\mathsf{T}_1, \mathsf{T}_3\}$. The basic framework will generate 16 key-value pairs. Suppose we aggregate them into three groups, $\mathcal{G}_1 = \{\mathsf{T}_1, \mathsf{T}_2\}, \mathcal{G}_2 = \{\mathsf{T}_1, \mathsf{T}_3\}, \mathcal{G}_3 = \{\mathsf{T}_1, \mathsf{T}_4\}$. 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., $\langle e, \{\mathcal{G}_1(e), \mathcal{G}_2(e), \ldots, \mathcal{G}_l(e)\}$. 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 $\langle \mathsf{C}, \{v_1, v_2, \ldots\}\rangle \rightarrow \langle v_i, \mathsf{C}\rangle$. $\langle e, \{\mathcal{G}_1(e), \ldots, \mathcal{G}_l(e)\}\rangle \rightarrow \langle e, \mathcal{G}_i(e)\rangle$.

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 $1 \leq i \leq l$.

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

Different from the basic framework, for each column, it outputs key-value pairs $\langle \mathsf{C}, list(\mathcal{G})\rangle$, 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 $\langle \mathsf{C}, list(\mathcal{G})\rangle \rightarrow \langle \mathsf{C}, list(\mathcal{G})\rangle$.

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

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

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 $\mathsf{T} \in \mathcal{G} \in list(list(\mathcal{G}))$. 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., $\mathcal{H}_T^{\mathcal{G}}(\mathcal{G})$. For each type in $\mathcal{H}_T^{\mathcal{G}}(\mathcal{G})$, 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 $\mathsf{topk}(\mathsf{T})$.

ReduceSetup To get $\mathcal{H}_T^{\mathcal{G}}$ in each reducer, we take $\mathcal{H}_T^{\mathcal{G}}$ 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.

Algorithm 2: Aggregation-based Algorithm

```
// the first stage
Map(⟨C/e, {v1, v2, ...}/{G1(e), G2(e), ...}⟩)
    foreach column ⟨C, {v1, v2, ...}⟩ do output(⟨vi, C⟩);
    foreach entity ⟨e, {G1(e), G2(e), ...}⟩ do
        output(⟨e, Gi(e)⟩);
Reduce(⟨v = e, list(C/G)⟩)
    foreach C/T in list(C/G) do
        if C/T is a group then add G to list(G);
        else if C/T is a column then add C to list(C);
    foreach C in list(C) do output(⟨C, list(G)⟩);
// the second stage
Map(⟨C, list(G)⟩)
    output(⟨C, list(G)⟩);
ReduceSetup()
    Each reducer loads side data H_T^G from DFSx;
Reduce(⟨C, list(list(G))⟩)
    Initialize a hash map H;
    foreach G in list(list(G)) do
        foreach T in H_T^G(G) do
            if T ∈ H then H(T) = H(T) + 1;
            else H(T) = 1;
    topk(T) = top-k types with largest H(T);
    output(⟨C, topk(T)⟩);
```

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 $\langle e \in \mathsf{T}, \mathsf{T}\rangle$ and $\langle c \in \mathsf{C}, \mathsf{C}\rangle$. The aggregation method counts their overlap based on $\langle e, \mathcal{G}_i(e)\rangle$ and $\langle c, \mathsf{C}\rangle$. As $\cup_{1 \ i \ l}\mathcal{G}_i(e) = \mathcal{L}(e)$, and $\mathcal{G}_i(e) \cap \mathcal{G}_j(e) = \phi$ for $i \neq j$, the two counted overlap numbers are the same. Thus the aggregation based method satisfies correctness. □

Example 3. *Consider three groups $\mathcal{G}_1 = \{\mathsf{T}_1, \mathsf{T}_2\}, \mathcal{G}_2 = \{\mathsf{T}_1, \mathsf{T}_3\}, \mathcal{G}_3 = \{\mathsf{T}_1, \mathsf{T}_4\}$. 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 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 $\mathcal{H}_T^{\mathcal{G}}$ to compute overlap between each type and column. For example, consider $\langle \mathsf{C}_1, \{\{\mathcal{G}_1\}, \{\mathcal{G}_1\}, \{\mathcal{G}_2\}, \{\mathcal{G}_2\}\}\rangle$. Based on $\mathcal{G}_1 = \{\mathsf{T}_1, \mathsf{T}_2\}, \mathcal{G}_2 = \{\mathsf{T}_1, \mathsf{T}_3\}$, we have the overlap of T_1 with C_1 is 4, and the overlap of T_2 (T_3) with column C_1 is 2 (2). Thus the top-1 type of column C_1 is T_1.*

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.

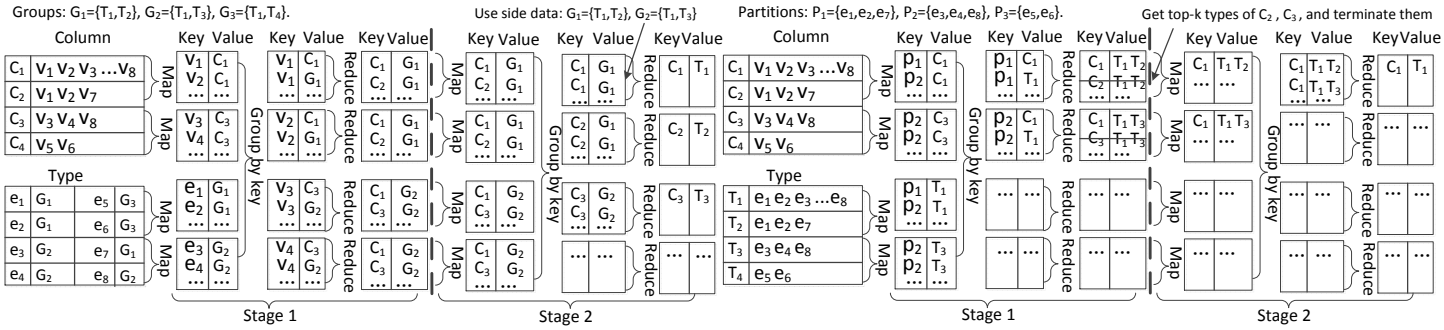

Figure 4: Knowledge type aggregation based method.

Figure 5: Knowledge entity partition based method.

The aggregation method includes two parts: (1) using the groups to generate the key-value pairs for each entity; and (2) distributing the groups $\mathcal{H}_T^{\mathcal{G}}$ 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.

5.2 Aggregation Cost Model

The basic framework generates $\langle e, \mathsf{T}\rangle$ key-value pairs. The aggregation based method reduces the number of key-value pairs by aggregating $\langle e, \mathsf{T}\rangle$ 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 $\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_{j\mathcal{G}j}$ denote the union of groups of all entities. Let E_i denote the set of entities whose groups include $\mathcal{G}_i$. That is $\mathsf{E}_i = \{e | \exists j, \mathcal{G}_i = \mathcal{G}_j(e)\}$. The aggregation based method distributes groups $\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_{j\mathcal{G}j}$ as the side data. For each entity $e \in \mathsf{E}_i$, 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

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

and the total cost of the basic framework is

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

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 can prove the Theorem by a reduction from the 3-CNF-SAT problem. Due to space constraints, we omit the proof and interested readers refer to the full paper [z] for more details. □

[z] http://dbgroup.cs.tsinghua.edu.cn/ligl/ccd.pdf

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

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

Based on this observation, we devise a greedy algorithm. For each entity e, we enumerate all possible groups on $\mathcal{L}(e)$. (Since the number of types that contain an entity is small, the number of possible groups 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 $\alpha|\mathsf{E}_i| \times |\mathcal{G}_i| - (\alpha|\mathsf{E}_i| + \beta|\mathcal{G}_i|)$. 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 T with no subtype. Let $\mathcal{G}_i$ denote the set of types that have a directed path to T (including itself). Obviously any type in $\mathcal{G}_i$ will take T as its subtype or has a descendant which takes T as a subtype. Let E denote the entity set of T. We take $\mathcal{G}_i$ as a group. Then we remove the type T from the knowledge base and remove the entities in 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 $\langle \mathcal{G}_i, \mathcal{H}_T^{\mathcal{G}}(\mathcal{G}_i)\rangle$ for the side data. We only need to slightly change our algorithm. For each pair $\langle \mathsf{E}, \mathcal{G}_i\rangle$, we generate pair $\langle \mathcal{G}_i, \mathcal{H}_T^{\mathcal{G}}(\mathcal{G}_i)\rangle$. For each entity e in 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 $\mathsf{T}_2, \mathsf{T}_3, \mathsf{T}_4$ are subtypes of type T_1. Thus we can aggregate T_2 with T_1 and generate a group $\{\mathsf{T}_1, \mathsf{T}_2\}$ which will be inserted into the side data $\mathcal{H}_T^{\mathcal{G}}$. As T_2 contains $\{e_1, e_2, e_7\}$, entities e_1, e_2, e_7 will use the group $\{\mathsf{T}_1, \mathsf{T}_2\}$ and we will add the group into the MapReduce file.

Notice that our techniques do not dependent on type hierarchies, i.e., our methods still work if there is no type hierar-

chy. On the other hand, we utilize type hierarchies heavily to improve the performance and scalability. We can easily extract hierarchies from YAGO and Freebase. For example, we can use the *included types*$^{\times}$ associated with each Freebase type to construct a rich 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.)

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 $\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_p$. If all the cell values in a column fall in the same partition, e.g., P_i, we only use this partition to annotate the column and prune other unnecessary partitions. This method not only prunes many unnecessary partitions,