Schemaless Join for Result Set Preferences

Chuancong Gao
Simon Fraser University
cga31@sfu.ca

Jian Pei
Simon Fraser University
Huawei Technologies Co., Ltd.
jpei@cs.sfu.ca

Jiannan Wang
Simon Fraser University
jnwang@sfu.ca

Yi Chang
Huawei Research America
yichang@acm.org

***Abstract*—In many applications, such as data integration and big data analytics, one has to integrate data from multiple sources without detailed and accurate schema information. The state of the art focuses on matching attributes among sources based on the information derived from the data in those sources. However, a best join result according to a method's own pre-determined criteria may not fit a user's best interest.**

In this paper, we tackle the challenge from a novel angle and investigate how to join schemaless tables to meet a user preference the best. We identify a set of essential preferences that are useful in various scenarios, such as minimizing the number of tuples in outer join results and maximizing the entropy of the joining key's distribution. We also develop a systematic method to compute the best join predicate optimizing an objective function representing a user preference. We conduct extensive experiments on 4 large datasets and compare with 4 baselines from the state of the art of schema matching and attribute clustering. The experimental results clearly show that our algorithm outperforms the baselines significantly in accuracy in all the cases, and consumes comparable running time.

I. Introduction

In many applications, one has to integrate data from multiple sources. More often than not, a data source can be modeled as a relational table. Due to various reasons, the schema information about those data sources may not be accurate, complete, or even available. For one example, different sources may have incompatible naming standard for attribute names. For another example, web tables crawled online often have no schema. Furthermore, attribute constraints (like primary/foreign-key constraint) often do not exist among different sources. The needs of integrating multiple data sources with incomplete or even missing meta data have strongly motivated the fruitful research on schema matching [13], [12], [7], [19], [11] and attribute clustering [6], [24], [23], [24]. While we will review the related work in Section II, the state of the art focuses on matching attributes among sources using the information derived from the data in the tables to be joined. However, a best join result according to a method's own pre-determined criteria may not fit a user's best interest. As we know, a primary/foreign-key constraint explicitly defines how two tables should be joined. However, when the constraint is not applicable, it is possible to have multiple ways to join the data, and different users may expect different join results.

This work is supported in part by the NSERC Discovery Grant program, the Canada Research Chair program, the NSERC Strategic Grant program, and the SFU President's Research Start-up Grant. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

ID	Tel.	Name	Home	Clerk
P1	001	Alice	Main St.	Steve
P2	002	Alice	Royal Rd.	Steve
P3	003	Bob	Robson St.	Ada
P4	004	Lucy	Univ Dr.	Steve
P5	005	Steve	Main St.	Steve
P6	006	Tom	Beta Ave.	Steve

R (People)

ID	Client	Address	Tel.	Date	Manager
O1	Alice	Main St.	001	Apr 1	Steve
O2	Bob	Royal Rd.	010	Apr 1	Alice
O3	Bob	Robson St.	003	Apr 2	Steve
O4	Alice	Main St.	001	Apr 3	Steve
O5	Tom	Beta Ave.	006	Apr 4	Steve
O6	Peter	Main St.	011	Apr 5	Alice

S (Orders)

Fig. 1: A toy example of two tables.

In this paper, we tackle this data integration challenge by including human in the loop – we investigate how to join tables guided by user preferences. It is well known that user guidance may help to improve data integration quality [21], [22]. More importantly, allowing users to specify preferences on join results can substantially improve user experience.

From a user's point of view, she only needs to understand and specify the preference over the potential join results at the macro-level, and the best join result w.r.t. the preference is computed without user interaction at the micro-level.

Consider a toy example of tables R and S in Figure 1. Suppose the attribute information as well as the primary/foreign-key information is unavailable. Only the data in the two tables is available. For the ease of discussion, in Figure 1 we add some column names so that our discussion is easy to follow. Such column names are unavailable to our algorithms.

One may guess table R may contain information about customers, and S may be about the orders of a product. In one scenario, the company may want to send out advertisements to as many customers who purchase the product in the past. Then, the company would like to join the two tables in the way that the most tuples in S are matched. To meet this preference, as shown in Figure 2, joining column "Home" in R and column "Address" in S gives the most desirable result. In another situation, the company may want to obtain purchase information with the least ambiguous customer information. That is, the company wants to match each order record with fewest customer records. To meet this preference, as shown in Figure 2, joining the "Tel." column in R with the "Tel." column in S, the "Name" column in R with the "Client" column in S, and the "Home" column in R with the "Address" column in S produces the most preferable answer.

As illustrated, under different preferences there are different ways to join tables. Can we formulate a set of essential preferences that users can use to specify their preferences in

many application scenarios? Moreover, can we compute the best way to join tables so that the most desirable results can be produced to echo user preferences? Motivated by the above questions, in this paper, we formulate and tackle the problem of schemaless join for result set preferences. To the best of our knowledge, this problem has not been touched in literature. We make a few contributions.

First, we formulate result set preferences and the problem of schemaless join for result set preferences. We investigate the monotonicity of preferences that is very useful in algorithm development. Second, we devise a general predicate enumeration algorithm framework for the problem, and develop several critical pruning techniques. Last, we conduct an extensive empirical study on 4 large datasets and compare with 4 state of the art baselines from schema matching and attribute clustering. The results clearly show that our algorithm is effective and efficient.

The rest of the paper is organized as follows. We review the related work in Section II. We present our problem formulation in Section III, discuss the preferences in Section IV, and develop our algorithm in Section V. We report an empirical study in Section VI, and conclude the paper in Section VII.

II. Related Work

In this section, we review the previous work on schema matching and schemaless join, attribute clustering, and foreign-key discovery. There are studies about preference in database queries (see [18], [8] for thorough surveys). However, they mainly focus on preferences where users explicitly specify both the querying condition and the respective ordering of results against some specific types of data For example, for keyword search [17], preference $\{\{\text{sci-fi}\}, \text{alien} > \text{predator}\}$ indicates that the records related to "alien" are more preferred than the records related to "predator" when searching for "sci-fi". Thus, we do not review those methods here. To the best of our knowledge, [10] is the only work discussing result set preference for joining similar objects.

A. Schema Matching and Schemaless Join

Schema matching and schemaless join are highly related. However, schema matching focuses on aggressively matching as many semantically equivalent attributes as possible for compact integrated results among different sources , while schemaless join focuses on joining only essential attributes for accurate join results. Rahm and Bernstein [13] surveyed schema matching, where the existing work are categorized into pure schema-based, data instance-based, and external knowledge-based methods. As our method does not rely on any schema or external knowledge, we only discuss the data instance-based methods, where only the data instances in the tables are considered.

Kang and Naughton [12] computed the entropy of each column and the mutual information between columns of the same table, for building a dependency graph of each table. The best attribute mappings minimize the distance between two dependency graphs according to the proposed distance metric. By finding duplicate values in a dataset, Bilke and Naumann [7] found mapping between attributes in different tables. Warren and Tompa [19] conducted schema mappings by finding string transformation rules between attributes. Acar and Motro [5] found the mostly likely join plan by finding a spanning tree among schemaless tables. Edges denoting candidate attribute pairs are filtered out by Jaccard similarity among attributes and weighted by information gain. Because of spanning tree, joining two tables with more than one attribute pairs are not fully supported. Hassanzadeh *et al.* [11] found linkage points on Web data, where each linkage point is a single matching attribute pair. Different set similarities, such as Jaccard similarity, are used to measure the similarity between the values in two attributes.

The existing works of both schema matching and schemaless join compute the best result according to their own criteria. However, the "best" result does not always match a user's best interest. In this paper, we join the columns according to a user specified preference, where each preference is devised to try to capture a specific aspect of the user intent.

B. Foreign-Key Discovery

Given the primary-key of each relation, foreign-key discovery tries to find the unknown foreign-keys among different relations to join with each primary-key. While schemaless join does not require any constraint among attributes, foreign-key discovery assumes the existence of a hidden primary/foreign-key constraint which represents the only correct way to join the data. Most of the methods assume the knowledge about primary-key columns. The existing methods like GORDIAN [16] computes the primary-key columns of a table by efficiently encoding the tuples into a prefix tree. However, there are usually many primary-key column combinations and the actual primary-key cannot always be identified accurately.

Adopting machine learning techniques, Rostin *et al.* [14] found all foreign-key constraints in a database, by computing all the inclusive dependencies between columns and classifying the relationship with 10 manually crafted features. Using the earth mover distance, Zhang *et al.* [23] put a primary-key column and its corresponding foreign-key columns into the same cluster, and found all foreign-key columns. [23] is also capable of finding multi-column foreign-keys. Recently, [9] proposed a fast way for primary/foreign-key discovery. Unlike other work, this work does not require knowing the primary-key. Instead, all the candidate attribute pairs satisfying the primary/foreign-key constraint are scored by a combined scoring function. Then, the method finds the join paths to connect the tables using the currently highest scoring candidate. [20] also finds the foreign-keys by machine learning, but in an unsupervised way. It returns the most similar columns to the primary-key as the foreign-keys, by measuring the distance using in total 6 features.

C. Attribute Clustering

Schemaless join works on joining attributes sharing the same semantic meaning. Instead, attribute clustering groups

columns from a set of tables into multiple clusters, where the columns in a cluster share a common semantic meaning. However, attribute clustering cannot tell which cluster of attributes should participate in the join process.

Ahmadi *et al.* [6] categorized columns based on their data types. The method builds different signatures for categorizing attributes based on the types of semantics inferred from the data in attributes, such as phone numbers and email addresses. For each category, the most frequent q-grams are used to build the signatures. Zhang *et al.* [24] grouped columns into clusters according to the relationship either primary/foreign-key columns, columns from different views of the same base table, or semantically equivalent columns like customer name from different tables. Their method used the earth mover distance to correlate columns [23]. In [24], a database is interpreted as a graph where nodes represent database columns and edges represent column relationships. The graph is further decomposed into connected components and clusters.

III. Problem Definition

Consider two tables R and S. Denote by $\mathcal{C}^R = \{c_1^R, c_2^R, \ldots, c_m^R\}$ and $\mathcal{C}^S = \{c_1^S, c_2^S, \ldots, c_n^S\}$ the sets of attributes in R and S, respectively. We assume that **the schema about the data is unavailable**. That is, for each table, the relation schema, which defines the attribute names and domain constraints, is unavailable. Moreover, the schema crossing relations, which defines the mapping constraints between attributes of different relations, is assumed unavailable, too. The data in each table is the only information that we use. Our assumption of no schema information addresses the challenges in many data cleaning and integration scenarios.

We consider equi-join in this paper. A join predicate specifies the matching attributes that are used to join two tables.

Definition 1: A **join predicate** p is a set of attribute pairs $\left\{\left(c_{i_1}^R, c_{j_1}^S\right), \left(c_{i_2}^R, c_{j_2}^S\right), \ldots, \left(c_{i_{|p|}}^R, c_{j_{|p|}}^S\right)\right\}$, where $c_{i_x}^R \in \mathcal{C}^R$ $(1 \le i_x \le m)$ and $c_{j_x}^S \in \mathcal{C}^S$ $(1 \le j_x \le n)$. ■

For example, consider the tables in Figure 1. Predicate $\{(\text{Name}, \text{Client})\}$ specifies joining column "Name" in R and column "Client" in S.

The join result has all the $m+n$ attributes from R and S. Specially, empty join predicate $\emptyset$ leads to a Cartesian product of two tables.

The **inner join** on predicate p is

$$R \bowtie_p S = \sigma_{c_{i_1}^R = c_{j_1}^S \wedge c_{i_2}^R = c_{j_2}^S \wedge \ldots \wedge c_{i_{|p|}}^R = c_{j_{|p|}}^S}(R \times S)$$

Figure 2 gives a few examples of inner join using various predicates on the tables in Figure 1.

We further define outer join. Denote by $(null, \ldots)_n$ an n-tuple of $null$ values. Then, the **full outer join** is

$$\begin{aligned} R \bowtie_p^{\leftrightarrow} S = {} & (R \bowtie_p S) \\ & \cup \left(\left(R \setminus \pi_{c_1^R, c_2^R, \ldots, c_m^R}(R \bowtie_p S)\right) \times \{(null, \ldots)_n\}\right) \\ & \cup \left(\{(null, \ldots)_m\} \times \left(S \setminus \pi_{c_1^S, c_2^S, \ldots, c_n^S}(R \bowtie_p S)\right)\right) \end{aligned}$$

For example, in Figure 2, for $p_2 = \{(\text{Tel.}, \text{Tel.}), (\text{Name}, \text{Client}), (\text{Home}, \text{Address})\}$, comparing to $R \bowtie_{p_2} S$, $R \bowtie_{p_2}^{\leftrightarrow} S$ contains three additional tuples, resulted from joining three tuples in R of IDs P2, P4, and P5 with $(null, \ldots)_n$, and another two additional tuples, resulted from joining $(null, \ldots)_m$ with two tuples in S of IDs O2 and O6.

Immediately, we have the following property.

Property 1: For two predicates p and q such that $p \subset q$, $R \bowtie_p S \supseteq R \bowtie_q S$. ■

In general, we have the following result.

Theorem 1: Let p and q be two predicates. If $R \bowtie_p S = R \bowtie_q S$, then $R \bowtie_{p \cup q} S = R \bowtie_p S = R \bowtie_q S$.

Proof: Due to Property 1, $R \bowtie_p S \supseteq R \bowtie_{p \cup q} S$. We only need to show $R \bowtie_p S \subseteq R \bowtie_{p \cup q} S$. For any tuple $t \in R \bowtie_p S$ and any attribute pair $(c_i^R, c_j^S) \in p \cup q$, $\pi_{c_i^R}(t) = \pi_{c_j^S}(t)$. Thus, $t \in R \bowtie_{p \cup q} S$ and $R \bowtie_p S \subseteq R \bowtie_{p \cup q} S$. ■

Definition 2: A predicate p is **closed** if there does not exist another predicate $q \supset p$ where $R \bowtie_p S = R \bowtie_q S$. ■

For example, for the two tables in Figure 1, predicate $p_2' = \{(\text{Tel.}, \text{Tel.})\}$ is not closed, since $R \bowtie_{p_2'} S = R \bowtie_{p_2} S$, where $p_2 = \{(\text{Tel.}, \text{Tel.}), (\text{Name}, \text{Client}), (\text{Home}, \text{Address})\}$, as shown in Figure 2. It can be verified that p_2 is closed. Apparently, using Theorem 1, we have the following result.

Corollary 1: Let p be a predicate. There exists one and only one closed predicate $q \supseteq p$ such that $R \bowtie_p S = R \bowtie_q S$. ■

We can define a relation $\sim$ among all possible predicates: for predicates p and q, $p \sim q$ if $R \bowtie_p S = R \bowtie_q S$.

Theorem 2: Let $\mathbb{P}$ be the set of predicates. Consider the subset partial order $\subset$ on $\mathbb{P}$. Then, $\sim$ is an equivalence relation on $\mathbb{P}$. Moreover, for each equivalence class in the quotient $\mathbb{P}/\sim$, there exists a unique upper-bound in the class, which is a closed predicate.

Proof: The reflexivity and symmetry are trivial. The transitivity holds since $\forall p, q, r \in \mathbb{P}$, if $p \sim q$ and $q \sim r$, then $R \bowtie_p S = R \bowtie_q S$ and $R \bowtie_q S = R \bowtie_r S$. Thus, $R \bowtie_p S = R \bowtie_r S$, that is, $p \sim r$.

Let C be an equivalence class in $\mathbb{P}/\sim$, and p and q be two upper-bounds of C such that $p \neq q$. According to Theorem 1, $R \bowtie_{p \cup q} S = R \bowtie_p S = R \bowtie_q S$. That is, $p \sim (p \cup q)$ and thus $p \cup q \in C$. As $p \subset (p \cup q)$, it contradicts the assumption that p is an upper-bound.

If the upper-bound $p \in C$ is not closed, then there exists a predicate $q \supset p$ such that $R \bowtie_p S = R \bowtie_q S$. Thus, $q \in C$, which contradicts that p is the upper-bound. ■

Often, predicates producing empty join results are not interesting. Thus, we call a predicate p **valid** on R and S if $R \bowtie_p S \neq \emptyset$. Without specific mentioning, we are interested in valid predicates only.

For a predicate p and tables R and S, the **set of joining keys** of $R \bowtie_p S$ is the set of values used by the predicate p in the join, that is, $K^{R,S}(p) = \pi_{c_{i_1}^R, c_{i_2}^R, \ldots, c_{i_{|p|}}^R}(R \bowtie_p S)$, where $p = \left\{\left(c_{i_1}^R, c_{j_1}^S\right), \left(c_{i_2}^R, c_{j_2}^S\right), \ldots, \left(c_{i_{|p|}}^R, c_{j_{|p|}}^S\right)\right\}$. When R and S are clear from context, we just write $K(p)$ instead of $K^{R,S}(p)$. For each joining key $k = (v_1, v_2, \ldots, v_{|p|}) \in K(p)$, we define the corresponding **joining group** as a pair

ID	Tel.	**Name**	Home	Clerk	ID	**Client**	Address	Tel.	Date	Manager
P1	001	Alice	Main St.	Steve	O1	Alice	Main St.	001	Apr 1	Steve
P1	001	Alice	Main St.	Steve	O4	Alice	Main St.	001	Apr 3	Steve
P2	002	Alice	Royal Rd.	Steve	O1	Alice	Main St.	001	Apr 1	Steve
P2	002	Alice	Royal Rd.	Steve	O4	Alice	Main St.	001	Apr 3	Steve
P3	003	Bob	Robson St.	Ada	O2	Bob	Royal Rd.	010	Apr 1	Alice
P3	003	Bob	Robson St.	Ada	O3	Bob	Robson St.	003	Apr 2	Alice
P6	006	Tom	Beta Ave.	Steve	O5	Tom	Beta St.	006	Apr 4	Steve

$p_1 = \{(\text{Name}, \text{Client})\}$

ID	**Tel.**	**Name**	**Home**	Clerk	ID	**Client**	**Address**	**Tel.**	Date	Manager
P1	001	Alice	Main St.	Steve	O1	Alice	Main St.	001	Apr 1	Steve
P1	001	Alice	Main St.	Steve	O4	Alice	Main St.	001	Apr 3	Steve
P3	003	Bob	Robson St.	Ada	O3	Bob	Robson St.	003	Apr 2	Alice
P6	006	Tom	Beta Ave.	Steve	O5	Tom	Beta Ave.	006	Apr 4	Steve

$p_2 = \{(\text{Tel.}, \text{Tel.}), (\text{Name}, \text{Client}), (\text{Home}, \text{Address})\}$

ID	Tel.	Name	**Home**	Clerk	ID	Client	**Address**	Tel.	Date	Manager
P1	001	Alice	Main St.	Steve	O1	Alice	Main St.	001	Apr 1	Steve
P1	001	Alice	Main St.	Steve	O4	Alice	Main St.	001	Apr 3	Steve
P1	001	Alice	Main St.	Steve	O6	Peter	Main St.	011	Apr 5	Alice
P5	005	Steve	Main St.	Steve	O1	Alice	Main St.	001	Apr 1	Steve
P5	005	Steve	Main St.	Steve	O4	Alice	Main St.	001	Apr 3	Steve
P5	005	Steve	Main St.	Steve	O6	Peter	Main St.	011	Apr 5	Alice
P3	003	Bob	Robson St.	Ada	O3	Bob	Robson St.	003	Apr 2	Alice
P2	002	Alice	Royal Rd.	Steve	O2	Bob	Royal Rd.	010	Apr 1	Alice
P6	006	Tom	Beta Ave.	Steve	O5	Tom	Beta Ave.	006	Apr 4	Steve

$p_3 = \{(\text{Home}, \text{Address})\}$

ID	Tel.	**Name**	Home	Clerk	ID	Client	Address	Tel.	Date	**Manager**
P1	001	Alice	Main St.	Steve	O2	Bob	Royal Rd.	010	Apr 1	Alice
P1	001	Alice	Main St.	Steve	O6	Peter	Main St.	011	Apr 5	Alice
P2	002	Alice	Royal Rd.	Steve	O2	Bob	Royal Rd.	010	Apr 1	Alice
P2	002	Alice	Royal Rd.	Steve	O6	Peter	Main St.	011	Apr 5	Alice
P5	005	Steve	Main St.	Steve	O1	Alice	Main St.	001	Apr 1	Steve
P5	005	Steve	Main St.	Steve	O3	Bob	Robson St.	003	Apr 2	Steve
P5	005	Steve	Main St.	Steve	O4	Alice	Main St.	001	Apr 3	Steve
P5	005	Steve	Main St.	Steve	O5	Tom	Beta Ave.	006	Apr 4	Steve

$p_4 = \{(\text{Name}, \text{Manager})\}$

Fig. 2: Examples of $R \ltimes_{p_i} S$ using each predicate p_i.

$\left(G_k^R, G_k^S\right)$, where G_k^R and G_k^S are the sets of tuples in R and those in S, respectively, having the same key, that is, we have $G_k^R = \sigma_{c_{i_1}^R = v_1 \wedge c_{i_2}^R = v_2 \wedge \ldots \wedge c_{i_{|p|}}^R = v_{|p|}}(R)$ and $G_k^S = \sigma_{c_{j_1}^S = v_1 \wedge c_{j_2}^S = v_2 \wedge \ldots \wedge c_{j_{|p|}}^S = v_{|p|}}(S)$. Moreover, we denote by $r_k = \left|G_k^R\right|$ and $s_k = \left|G_k^S\right|$.

Take $p_2 = \{(\text{Tel.}, \text{Tel.}), (\text{Name}, \text{Client}), (\text{Home}, \text{Address})\}$ in Figure 2 as an example. $R \ltimes_{p_2} S$ has 3 joining keys, where $K(p_2) = \{(001, \text{Alice}, \text{Main St.}), (003, \text{Bob}, \text{Robson St.}), (006, \text{Tom}, \text{Beta Ave.})\}$. The first tuple in R and the first and the fourth tuples in S contains the first joining key $k_1 = (001, \text{Alice}, \text{Main St.})$. Thus, its joining group is $\left(G_{k_1}^R, G_{k_1}^S\right)$, where $G_{k_1}^R$ is a set of tuples of ID P1 and $G_{k_1}^S$ is a set of tuples of IDs O1 and O4.

In different application scenarios, users may prefer different join results. We will discuss a series of preferences on join results in Section IV. In general, we define a **preference scoring function** $h : (R, S, \mathbb{P}) \to \mathbb{R}$ to model a preference, where $\mathbb{P}$ is the set of all valid predicates on R and S, and $\mathbb{R}$ is the set of real numbers. The larger the preference value, the more preferable the predicate. For simplicity, we write $h(p)$ when R and S are clear from context.

Definition 3: A preference scoring function h is **monotonic**, if $h(p) \geq h(p')$ for any predicates p and p' where $p \subseteq p'$. ■

For example, let us consider a simple function $h_0(p) = |R \ltimes_p S|$. Obviously, for any predicates p and p' such that $p \subseteq p'$, $R \ltimes_p S \supseteq R \ltimes_{p'} S$ and thus $h_0(p) \geq h_0(p')$. Therefore, h_0 is monotonic.

Now, we are ready to state our problem.

Definition 4: Given two tables R and S and a preference scoring function h, find a valid and closed predicate p that maximizes $h(p)$. ■

IV. Preferences on Join Results

In this section we formulate a series of preferences on join results that capture different needs in various applications. Table I summarizes the preferences. Those preferences cover the most common scenarios.

A. Preference for More Tuples Joined

In our example in Figure 1, a task may prefer that as many customer records and order records as possible are matched (i.e., joined). This motivates our first preference that maximizes the number of tuples joined between R and S. Formally, we define preference scoring function

$$h_1(p) = \sum_{k \in K(p)} (r_k + s_k)$$

[11] adopts a threshold on schema matching coverage defined as $cov(p) = \frac{\sum_{k \in K(p)} (r_k + s_k)}{|R| + |S|}$. Since $|R| + |S|$ is a constant when R and S are given, h_1 and coverage capture the same preference.

B. Preference for More Joining Groups

In our example in Figure 1, when the two tables are joined, each joining group may possibly be interpreted as a customer (with different profiles) and the corresponding purchasing records, that is, the items that the same customer purchased. Thus, each joining key uniquely identifies a customer's profiles and her purchasing records. A possible preference may identify as many unique customers as possible. This is equivalent to maximizing the number of joining keys. As a special case, the joining keys are the values of foreign-key when primary/foreign-key constraint exists, where a larger cardinality of foreign-key values is preferred [14]. Formally, we define preference scoring function

$$h_2(p) = |K(p)|$$

C. Preference for Less Tuples in Outer Join

In our example in Figure 1, the company may want to merge the customer records and the purchasing records, and the result should include not only the matched records but also the unmatched records. The merged table is actually the full outer join result. Given different join predicates, the outer join results have different numbers of tuples. To reflect the need of

TABLE I: Summary of the preferences.

Preference Description	Scoring Function	Monotonic	Monotonic Upper-Bound Function
Maximizing #involved tuples (coverage)	$h_1(p) = \sum_{k \in K(p)} (r_k + s_k)$	✓	
Maximizing #joining groups	$h_2(p) = \lvert K(p) \rvert$		$\widehat{h_2}(p) = \min\left\{\sum_{k \in K(p)} r_k, \sum_{k \in K(p)} s_k\right\}$
Minimizing #full outer join tuples	$h_3(p) = \sum_{k \in K(p)} (r_k + s_k - r_k \cdot s_k)$		$\widehat{h_3}(p) = \min\left\{\sum_{k \in K(p)} r_k, \sum_{k \in K(p)} s_k\right\}$
Maximizing entropy of joining key's distribution	$h_4(p) = -\sum_{k \in K(p)} P(k) \log_2 P(k)$		$\widehat{h_4}(p) = \log_2 \min\left\{\sum_{k \in K(p)} r_k, \sum_{k \in K(p)} s_k\right\}$
Maximizing harmonic mean of coverage and strength	$h_5(p) = \frac{2 \cdot \sum_{k \in K(p)} (r_k + s_k)}{2 \cdot \lvert R \ltimes_p S \rvert + \lvert R \rvert + \lvert S \rvert}$		$\widehat{h_5}(p) = \frac{2 \cdot \sum_{k \in K(p)} (r_k + s_k)}{\sum_{k \in K(p)} (r_k + s_k) + \lvert R \rvert + \lvert S \rvert}$

the company, the user expects the most compact merged result. Thus, we consider a preference having the outer join result contain as less tuples as possible. To capture this preference, we define preference scoring function

$$h_3(p) = \sum_{k \in K(p)} (r_k + s_k - r_k \cdot s_k)$$

As $|R| + |S|$ is a constant when R and S are given, according to the definitions of inner and outer joins, we have $|R \ltimes_p S| = \sum_{k \in K(p)} r_k \cdot s_k$ and further $\arg\min_p \ (|R| + |S| - h_3(p)) = \arg\min_p \ \left|R \ltimes_p^{\leftrightarrow} S\right| = \arg\max_p \ h_3(p)$.

D. Preference for More Balanced Distribution of Joining Keys

Consider our example in Figure 1, let the customers be Internet service users, and the orders be their monthly bills in the past year. Although a customer may have multiple profiles, due to reasons like moving, she would still receive a bill every month by continuing the service, resulting in a fixed number of bills throughout the year. Thus, we expect most customers have similar numbers of matched records of customer profiles and orders. Recall that each joining key uniquely identifies each customer's profiles and orders. Thus, a more balanced distribution of the numbers of joined tuples w.r.t. joining keys is preferred. As a special case, when primary/foreign-key constraint exists, the values of foreign-key are the joining keys. [23] demonstrates that in most circumstances the values of foreign-key form (nearly) uniform distribution with high randomness, We propose a preference based on the entropy of the joining key probability distribution. As we know, the higher the entropy, the more balanced the number of joined tuples $r_k \cdot s_k$ for each joining key k. Formally, the preference scoring function is

$$h_4(p) = -\sum_{k \in K(p)} P(k) \log_2 P(k) \text{ where } P(k) = \frac{r_k \cdot s_k}{|R \ltimes_p S|}$$

E. Preference for Greater Mean of Coverage and Strength

Consider a task of the example in Figure 1. To reduce the uncertainty of which customer made each order, we may prefer each order matches with only one customer. In sum, we may prefer one-to-one / one-to-many matching to many-to-many matching, i.e. r_k and / or s_k being 1 for each joining group. Thus, each involved record should appear unique in the join result, which is measured by strength. It is introduced in [11] for better schema matching accuracy, and defined as $str(p) = \frac{\sum_{k \in K(p)} (r_k + s_k)}{2 \cdot |R \ltimes_p S|}$. Usually, the more attributes joined, the higher confidence we have on the matched records in the join result.

Algorithm 1: Algorithm framework.

Function: $main(R, S, h)$
Input: tables R and S, preference h
Output: valid closed predicate that maximizes h
1 $p^* \leftarrow \emptyset$ // the currently best predicate
2 $enumerate(\emptyset)$ // recursively update p^*
3 **return** p^*

Function: $enumerate(p)$
Input: current predicate p
1 **if** $\widehat{h}(p) < h(p^*)$ **or** $\exists p' <^{lex} p : p' \supset p \wedge |R \ltimes_{p'} S| = |R \ltimes_p S|$ **then**
2 **return** // p can be pruned
3 **if** $h(p) > h(p^*)$ **or** $h(p) = h(p^*) \wedge p \supset p^*$ **then**
4 $p^* \leftarrow p$ // update p^* with p
5 **foreach** *attribute pair* $(c_{l_i}^R, c_{w_j}^S)$ *by multi-way merge join on* $L_{l_i}^R$ *and* $L_{w_j}^S$ *for all the joining groups of* p **do**
6 $enumerate(p \cup (c_{l_i}^R, c_{w_j}^S))$ // recursively extend p

However, we do not want to risk preventing joining records with only partially matched information. Recall that coverage discussed earlier measures the number of involved (matched) records in the join result. To balance the situation, we use the harmonic mean of coverage and strength as a preference scoring function.

$$h_5(p) = \frac{2 \cdot cov(p) \cdot str(p)}{cov(p) + str(p)} = \frac{2 \cdot \sum_{k \in K(p)} (r_k + s_k)}{2 \cdot |R \ltimes_p S| + |R| + |S|}$$

V. Finding Best Predicates

Given a preference on result sets defined in Section IV, how can we efficiently find the best join predicate? We present our algorithm in this section. We start with a top-down predicate enumeration framework, and then discuss the details of the critical steps. Algorithm 1 is the pseudo-code of our method. Our algorithm can handle all preferences discussed in Section IV.

A. A Predicate Enumeration Framework

Our algorithm enumerates all possible valid predicates starting from the empty predicate $\emptyset$, which serves as the root of our search. The enumeration process follows a set enumeration tree [15]. Iteratively we try to add a new pair of attributes from the two tables to expand the parent predicate.

To enable the enumeration, we assume a total order on attributes across the tables. Without loss of generality, we use the order that the attributes are listed in a table (and R is before S). For two attributes a and b in table $\mathcal{C}^R$, we write $a < b$ if

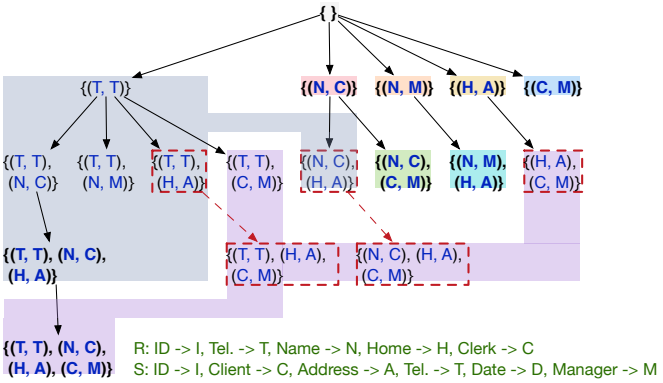

Fig. 3: The predicate enumeration tree of valid predicates. Predicates with the same background are in the same equivalence class over relation $\sim$. Predicates in bold are closed, and predicates surrounded by red dashed rectangles are pruned by the non-closed predicate pruning in Theorem 3.

a is ordered before b. For example, in Figure 1, we assume ID $<$ Name in R. Based on the total orders on attributes, we define a total order on attribute pairs. For two attribute pairs (a, b) and (a', b') , $(a, b) < (a', b')$ if (1) $a < a'$; or (2) $a = a'$ and $b < b'$. In a predicate, all attribute pairs are listed in the total order.

Now, we have a lexicographical order $<^{lex}$ on predicates, based on the total order on attribute pairs. For example, in Figure 1 $\{(\text{Tel.}, \text{Tel.})\} <^{lex} \{(\text{Tel.}, \text{Tel.}), (\text{Name}, \text{Client})\}$. Using the lexicographical order, all predicates can be enumerated in a set enumeration tree. The root of the tree is the empty set predicate $\emptyset$. Each node in the tree is a predicate $\{x_1, x_2, \ldots, x_i\}$ where all x_i's are in order. The children of the node are all predicates $\{x_1, x_2, \ldots, x_i, x_j\}$ such that $\{x_1, x_2, \ldots, x_i\} <^{lex} \{x_1, x_2, \ldots, x_i, x_j\}$. Figure 3 shows the set enumeration tree in our example in Figure 1.

As established by the set enumeration tree construction [15], each predicate appears in the set enumeration tree once and only once. Our algorithm framework conducts a depth-first search of the predicate set enumeration tree. For each predicate searched, we calculate the corresponding preference score. At the end of the search, our algorithm returns the predicate with the best preference score. The correctness of our algorithm is established immediately according to the above discussion.

B. Different Pruning Strategies

Not every predicate is valid or closed or maximizes the preference scoring function. Therefore, for the interest of efficiency, we devise the respective pruning techniques.

1) Invalid Predicate Pruning: Pruning invalid predicates is straightforward. According to Property 1, any superset of an invalid predicate cannot be valid. Therefore, once the search process encounters an invalid predicate p in the predicate enumeration tree, the whole subtree rooted at p can be pruned, as only the supersets of p appear in the subtree.

2) Non-Closed Predicate Pruning: To explore how to prune non-closed predicates, let us look at an example. Consider the tables in Figure 2 and the predicates in Figure 3. We can verify that predicate $p = \{(\text{Name}, \text{Client}), (\text{Home}, \text{Address})\}$ is not closed, as for predicate $p_2 = \{(\text{Tel.}, \text{Tel.})\} \cup p$, $R \ltimes_{p_2} S = R \ltimes_p S$. Therefore, every tuple in $R \ltimes_p S$ has the same value on the two "Tel." columns. For any predicate q that is a superset of p but does not contain attribute pair (Tel., Tel.), like $q = \{(\text{Name}, \text{Client}), (\text{Home}, \text{Address}), (\text{Clerk}, \text{Manager})\}$, we can always add (Tel., Tel.) to obtain another predicate $q' \supset q$ where $R \ltimes_q S = R \ltimes_{q'} S$. Thus, q cannot be closed.

We formulate the above observation as follows.

Theorem 3: If for predicates x and p, $|R \ltimes_p S| = |R \ltimes_{p \cup x} S|$. Then, for any predicate $q \supseteq p$ but $q \not\supset x$, q is not closed.

Proof: Let p and q be two predicates where $p \subseteq q$. According to Property 1, $R \ltimes_p S \supseteq R \ltimes_q S$. Thus, $R \ltimes_p S = R \ltimes_q S$ if and only if $|R \ltimes_p S| = |R \ltimes_q S|$.

Since $R \ltimes_p S = R \ltimes_{p \cup x} S$, every tuple in $R \ltimes_p S$ has the same values on attribute pairs $x \setminus p$, so does every tuple in $R \ltimes_q S$ on attribute pairs $x \setminus q$. Thus, $R \ltimes_q S = R \ltimes_{q \cup x} S$. As $q \not\supset x$, $q \subset q \cup x$. That is, q is not closed. ■

For predicate p, we check if there exists any previous enumerated predicate p' where $p' \supset p$ and $|R \ltimes_{p'} S| = |R \ltimes_p S|$. This can be done efficiently by memorizing previous enumerated predicates and sizes of their respective join results in a hash table, where each key is the size of a join result and its respective value is a list of predicates having the join results of the same size. We also remove any $p' \subset p$ in the hash table whenever $|R \ltimes_{p'} S| = |R \ltimes_p S|$ for more compact storage. The size of the hash table is bounded by the size of the enumeration tree. Because p' is enumerated before p, we have $p' <^{lex} p$ and further $q \not\supset p' \setminus p$ for any of p's descendant q. According to Theorem 3, p can be safely pruned. For example, in Figure 3, the whole subtree rooted at predicate $\{(\text{Name}, \text{Client}), (\text{Home}, \text{Address})\}$ can be pruned.

3) Monotonic Score-based Pruning: Since we are only interested in the predicate of the best preference score, we can use the monotonic preference scoring functions or monotonic upper-bound functions to prune unpromising predicates. For each node p in the predicate enumeration tree, we calculate the upper-bound of the preference scoring function at the node p. If the upper-bound is less than or equal to the best preference score obtained so far, the whole subtree rooted at p can be pruned due to the monotonicity of the upper-bound. We derive the monotonic upper-bounds of all our preferences as follows. Table I summarizes the preference upper-bounds.

Lemma 1: h_1 is monotonic.

Proof: Consider two predicates p and q such that $p \subset q$. Clearly, for any tuples $a \in R$ and $b \in S$, we have $a \times b \in R \ltimes_q S$ only if $a \times b \in R \ltimes_p S$. Since $\sum_{k \in K(p)} r_k = |\{a \in R \,|\, \exists b \in S : a \times b \in R \ltimes_p S\}|$, $\sum_{k \in K(p)} r_k \geq \sum_{k \in K(q)} r_k$. Thus, $\sum_{k \in K(p)} r_k$ is monotonic. Similarly, $\sum_{k \in K(p)} s_k$ is monotonic. Together, the lemma follows immediately. ■

h_2 is not monotonic. Consider predicate p_4 in Figure 2 and a new predicate $p_5 = \{(\text{Name}, \text{Manager}), (\text{Home}, \text{Address})\}$,

$p_4 \subset p_5$, as an example. However, $h_2(p_4) = 2 < h_2(p_5) = 3$. We have an upper-bound of h_2 that is monotonic.

Lemma 2: Let $\widehat{h_2}(p) = \min\left\{\sum_{k\in K(p)} r_k, \sum_{k\in K(p)} s_k\right\}$. Then, $h_2(p) \leq \widehat{h_2}(p)$ and $\widehat{h_2}(p)$ is monotonic.

Proof: For any joining key k, we must have $r_k \geq 1$ and further $|K(p)| \leq \sum_{k\in K(p)} r_k$. Similarly, $|K(p)| \leq \sum_{k\in K(p)} s_k$. Thus, $h_2(p) \leq \widehat{h_2}(p)$.

In the proof of Lemma 1, both $\sum_{k\in K(p)} r_k$ and $\sum_{k\in K(p)} s_k$ are monotonic. Thus, $\widehat{h_2}$ is monotonic. ■

h_3 is not monotonic, either. For example, in Figure 2, $p_3 \subset p_2$, but $h_3(p_3) = 2 < h_3(p_2) = 3$. h_2 and h_3 have the same monotonic upper-bound.

Lemma 3: $h_3(p) \leq \widehat{h_3}(p) = \widehat{h_2}(p)$.

Proof: As $r_k \geq 1$ and $r_k \geq 1$ for any joining key k, it is straightforward to get $\sum_{k\in K(p)} r_k \cdot s_k \geq \max\left\{\sum_{k\in K(p)} r_k, \sum_{k\in K(p)} s_k\right\}$. Thus, we have $h_3(p) \leq \sum_{k\in K(p)} (r_k + s_k) - \max\left\{\sum_{k\in K(p)} r_k, \sum_{k\in K(p)} s_k\right\} = \min\left\{\sum_{k\in K(p)} r_k, \sum_{k\in K(p)} s_k\right\} = \widehat{h_3}(p)$. ■

h_4 is not monotonic. In our example in Figure 1, $p_3 \subset p_2$, $h_4(p_3) = 1.447 < h_4(p_2) = 1.5$. As $h_4(p)$ is entropy, $\forall p : h_4(p) \leq \log_2 |K(p)| = \log_2 h_2(p)$. Thus, we have an upper-bound $\widehat{h_4}(p) = \log_2 \widehat{h_2}(p)$ for h_4. As $\widehat{h_2}(p)$ is monotonic, $\widehat{h_4}(p)$ is also monotonic.

h_5 is not monotonic. In our example in Figure 1, $p_1 \subset p_2$, $h_5(p_1) = 0.692 < h_5(p_2) = 0.7$.

Lemma 4: Let $\widehat{h_5}(p) = \frac{2\cdot\sum_{k\in K(p)}(r_k+s_k)}{\sum_{k\in K(p)}(r_k+s_k)+|R|+|S|}$. Then, $h_5(p) \leq \widehat{h_5}(p)$ and $\widehat{h_5}(p)$ is monotonic.

Proof: As $|R \bowtie_p S| \geq \sum_{k\in K(p)} r_k$ and $|R \bowtie_p S| \geq \sum_{k\in K(p)} s_k$, $h_5(p) \leq \widehat{h_5}(p)$. $\widehat{h_5}(p)$ is monotonic, as $\sum_{k\in K(p)} (r_k + s_k)$ is monotonic in Lemma 1. ■

C. Producing Children Nodes using Multi-Way Merge Joins

Let $p = \left\{\left(c_{i_1}^R, c_{j_1}^S\right), \left(c_{i_2}^R, c_{j_2}^S\right), \ldots, \left(c_{i_{|p|}}^R, c_{j_{|p|}}^S\right)\right\}$ be any predicate in the predicate set enumeration tree. A central step in our algorithm is to produce the children nodes of p. A brute-force method can enumerate every pair of attributes (c_i^R, c_i^S) such that $p <^{lex} p \cup \{(c_i^R, c_i^S)\}$, and compute the join result using the new predicate $p \cup \{(c_i^R, c_i^S)\}$. This is very costly in time since the number of possible pairs (c_i^R, c_i^S) and thus the complexity is proportional to $m \cdot n$ where m and n are the numbers of attributes in R and S, respectively. In this subsection, we develop a multi-way merge join technique that can reduce this cost substantially.

Using the current predicate p, the join result is a set of joining groups (G_k^R, G_k^S), where k is a joining key on the attributes in predicate p. When a new pair of attributes (c_i^R, c_i^S) is added to p, essentially each joining group is split into multiple joining groups for predicate $p \cup \{(c_i^R, c_i^S)\}$. Obviously, due to Property 1, no joining groups are merged.

Based on the above observation, we focus on how to split a single joining group into the joining groups corresponding to various children predicates of p. According to the lexicographical order, for table R we only need to consider those attributes not before the last attribute in p in the total order of attributes on R, and, for table S, we need to consider all those attributes. Let k be the number of attributes in R before the last attribute in p according to the total order $<$ on R. The number of attributes in R to be considered is $m - k$. For the sake of simplicity, let $c_{l_1}^R, \ldots, c_{l_{m-k}}^R$ be the attributes in table R that do not appear before the largest one in p, and $c_{w_1}^S, \ldots, c_{w_n}^S$ be the attributes in table S. For each $c_{l_i}^R$ $(1 \leq i \leq m-k)$, we sort all the tuples R in the joining group according to their values in attribute $c_{l_i}^R$, and denote the sorted list by $L_{l_i}^R$. Similarly we generate the sorted lists $L_{w_i}^S$. Here we assume there is an order on the possible values on all attributes, for example, the dictionary order.

Next, we conduct a multi-way merge join on the all the sorted lists. We consider the first value in each of those sorted lists. If the first values of two lists $L_{l_i}^R$ and $L_{w_j}^S$ match, then the tuples in G_k^R having this value on attribute $c_{l_i}^R$ and the tuples in G_k^S having this value on attribute $c_{w_j}^S$ form a new joining group. After the matching is conducted, those matched values and their corresponding tuples in the sorted lists are removed. If no match is found in the current round, the smallest value of the top values in those sorted lists and the corresponding tuples are removed.

For example, consider producing the children nodes of the predicate $\{(\text{Name}, \text{Client})\}$ in Figure 2. The candidate attributes are three attributes "Name", "Home", and "Clerk" in R and all the attributes in S. It has three different joining groups, and we conduct three multi-way merge joins respectively, stacking the new joining groups together according to the new predicates. Figure 4 gives an example of the second joining group, which has the joining key $k = \text{Bob}$ and contains tuples P3 in R and O2 and O3 in S. When splitting the joining group, we get the sorted list L_{Home}^R containing P3 with value "Robson St.", w.r.t. attribute "Home" in R. Similarly, we get the sorted list L_{Address}^S containing O3 with value "Robson St." and O2 with value "Royal Rd.", w.r.t. attribute "Address" in S. During the multi-way merge join, P3 in L_{Home}^R is matched with O3 in L_{Address}^S due to the same value "Robson St.", and they form a new joining group for the extending attribute pair $(\text{Home}, \text{Address})$ with the new joining key $k' = (\text{Bob}, \text{Robson St.})$.

Using the above method, we can scan each tuple once and produce all children nodes for the current predicate.

D. Generalization for More Than Two Tables

Limited by space, we briefly discuss how to generalize our preferences and algorithm for more than two tables. Suppose there are d tables $R_1, R_2, \ldots, R_d$. The generalized predicate p is a set of attribute pairs, where for each pair $\left(c_i^{R_x}, c_j^{R_y}\right)$ we have $c_i^{R_x} < c_j^{R_y}$, based on a total order of attributes across the tables. For example, $\left\{\left(c_2^{R_1}, c_1^{R_2}\right), \left(c_2^{R_2}, c_1^{R_3}\right)\right\}$ joins the second column of R_1 with the first column of R_2 and the second column of R_2 with the first column of R_3.

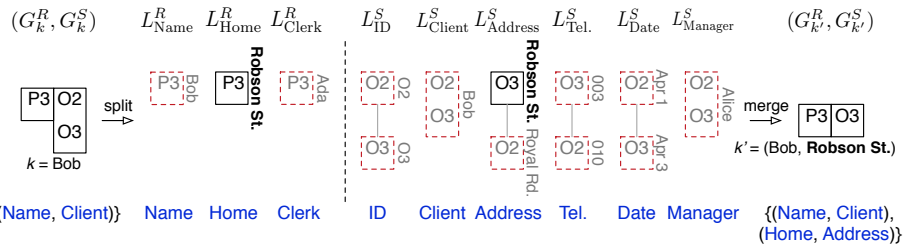

Fig. 4: Example of multi-way merge join for splitting the second joining group of predicate $\{(\text{Name}, \text{Client})\}$. Values in bold are matched. For each tuple, we only show its ID.

TABLE II: Dataset characteristics.

Dataset	Size	#Tables	#Col. per Table		#Tuples per Table		#Truths
			Average	Maximum	Average	Maximum	
TPC-H	1.0GB	8	7.6	16	1,082,656	6,001,215	8
Employees	140MB	6	4	6	653,169	2,844,047	6
IMDb	4.5GB	18	5.4	12	4,487,832	42,964,606	19
Yelp	1.0GB	5	62.8	170	371,067	1,125,458	5

The definitions of inner join and full outer join can be naturally generalized. The joining keys $K(p)$ are a set of values for tuples in inner join result on all attributes $c_i^{R_x}$. For each joining key $k \in K(p)$, the corresponding joining group is a tuple $\left(G_k^{R_1}, G_k^{R_2}, \ldots, G_k^{R_d}\right)$, where each $G_k^{R_m}$ $(1 \leq m \leq d)$ is the tuples in the m-th table R_m, respectively, having the same values on all attributes $c_i^{R_x} \in \mathcal{C}^{R_m}$ and $c_j^{R_y} \in \mathcal{C}^{R_m}$. Obviously, the previous definitions for two tables are just special cases here.

When joining more than two tables, a tuple may appear in more than one joining groups. For example, consider predicate $p_1 = \{(\text{Name}, \text{Client})\}$ in Figure 2. If R is also joined with another table on "Home" attribute, the first joining group may be further split into two joining groups. However, for S, both two new joining groups contain tuples O1 and O4. The first preference (maximizing #involved tuples) is now generalized to $h_1(p) = \sum_{1 \leq i \leq d} \left| \bigcup_{k \in K(p)} G_k^{R_i} \right|$. It is similar to generalize rest of the preferences based on the generalized definitions.

Based on the lexicographical order on generalized predicate, the current top-down enumeration framework and pruning strategies can still be adopted.

VI. Experiments

In this section, we report an extensive empirical study. The algorithms were implemented in Python and executed using PyPy [1] on a Mac Pro server with an Intel Xeon 3.70GHz CPU and 16GB memory, running OS X El Capitan.

We evaluate the methods using 4 datasets with ground-truths. Table II shows their statistical information.

- The standard benchmark dataset TPC-H [1], which has about 8 million records in total in 8 tables.
- A smaller sample dataset Employees [2] from MySQL, containing about 4 million records in total across 6 tables.
- IMDb [3] [2] is a challenging dataset, where every table contains a primary-key ID column and several foreign-key ID columns, all in the same domain. This dataset has about 80 million records in total across 18 tables.
- The real dataset from Yelp [4] is in JSON format with many nested properties. Each record is flattened by removing the nested levels. There are hundreds of properties, and the respective converted table contains at most 170 columns. The dataset has nearly 2 million records in total across 5 tables.

We compare our algorithm with four state of the art baselines. The first baseline "MIMatch" [12] computes schema mapping between two tables. Each attribute in R is either unmatched or uniquely matched with an attribute in S. As the algorithm can only do partial mapping (where not every attribute is matched) when using their normal distance metric, which does not support pruning, we set a predicate size limit of 4 to help it finish. We choose a control factor of $\alpha = 10$ for the normal distance metric, which in most cases gives the best results in terms of accuracy.

The second baseline "AttrCluster" [24] clusters columns from a set of tables into groups sharing the semantically related meanings. As a cluster may contain more than one column from R or S, we assume one column of R joins with another column of S in the same cluster. For example, clusters $\{c_1^R, c_2^R, c_1^S\}$ and $\{c_3^R, c_2^S\}$ are converted to two predicates $\{(c_1^R, c_1^S), (c_2^R, c_1^S)\}$ and $\{(c_3^R, c_2^S)\}$, where each predicate represents joining the columns in the same cluster sharing the same semantic meaning. We choose a threshold $\theta = 0.1$ for clustering, which shows the best results. It also requires a linear programming solver [3].

The third baseline "SMaSh" [11] finds a set of matching attribute pairs for schema matching. Each pair is evaluated individually by the algorithm. We choose Jaccard to measure

[1] PyPy (http://pypy.org/) is an advanced just-in-time compiler, having 10 times faster performance for our algorithm.

[2] It is converted to tables by using IMDbPy (http://imdbpy.sourceforge.net/).

[3] We use GNU Linear Programming Kit (https://www.gnu.org/software/glpk/) as the solver.

TABLE III: Average F_1-scores.

Dataset	Ours					MIMatch	AttrCluster	SMaSh		FastFK
	h_1	h_2	h_3	h_4	h_5			w/o F.	w/ F.	
TPC-H	0.958	**1**	**1**	**1**	**1**	0.167	0.696	0.938	0.875	0.875
Employees	0.833	**1**	**1**	**1**	0.833	0.306	0.667	0.667	0.833	0.833
IMDb	**0.632**	0.158	0.158	0	**0.632**	0.035	0.135	0.316	0.316	0.474
Yelp	0.8	**1**	**1**	**1**	**1**	n/a	0.45	0.733	0.8	0.8

the similarity between the sets of values from two attributes due to its best performance. SMaSh also proposes two measures, strength and coverage, for optional post-filtering of the candidate matching attribute pairs. As with and without filtering can lead to significantly different results, we use both as two baselines. According to [11], we set the set similarity threshold to 0.5, the strength threshold to 0.5, and the coverage threshold to 0.001.

The last baseline "FastFK" [9] is about foreign-key discovery. To the best of our knowledge, it is the only work that does not require the knowledge of primary-key. However, it also has a limitation that only single-column foreign-key can be returned. We set the parameter for containment constraint pruning $\epsilon = 0.1$ for the best results.

Both the baselines and our algorithm here work under the assumption that the schema is unavailable, and do not rely on the existence of external domain knowledge.

A. Accuracy

We want to evaluate how well each preference captures the nature of how tables are joined. However, due to the variety of user intent, it is difficult to measure the accuracy. We observe that the primary/foreign-keys, specified when designing a database's schema for traditional schema-based join, explicitly defines how the tables should be joined together. Thus, we use the known primary keys and their respective foreign keys in those data sets as the ground-truths, and evaluate how well using the preferences may approach the ground-truths. For each dataset, among all pairs of tables, those with primary/foreign-key relationship in the database schema are evaluated. The mapping of attributes between the primary-key in one table and its foreign-key in another table forms a ground-truth predicate of how the two tables are joined. Some databases have ground-truth predicates with more than one attribute pairs, like TPC-H and IMDb.

As both the ground-truth predicate and the output predicate may have more than one attribute pair, we use F_1-score, which is the harmonic mean of precision and recall, to measure the accuracy. When joining table R with S, if the output predicate is p and the ground-truth predicate is q, the precision and recall are defined as $\frac{|p \cap q|}{|p|}$ and $\frac{|p \cap q|}{|q|}$, respectively. For example, given output predicate $p = \left\{\left(c_1^R, c_1^S\right), \left(c_2^R, c_3^S\right)\right\}$ and ground-truth predicate $q = \left\{\left(c_2^R, c_3^S\right)\right\}$, $precision(p, q) = 0.5$ and $recall(p, q) = 1.0$. Thus, $F_1(p, q) = 0.667$. For the AttrCluster baseline which returns multiple predicates, we use the average F_1-score of all its predicates. There are multiple pairs of joining tables in each dataset. Thus, we report the average F_1-score.

Table III compares the 5 preferences on the 4 datasets. As the ground-truths are primary/foreign-keys in our experiment setting, the closer a preference reflects the primary/foreign-key relationship, the better its accuracy. h_1 (maximizing #involved tuples) simply maximizes the number of involved tuples. h_2 (maximizing #joining groups) and h_4 (maximizing distribution entropy) directly capture the characteristics of large cardinality and high randomness of primary/foreign-key [14], respectively. h_3 (minimizing #outer join tuples) and h_5 (maximizing harmonic mean of strength and coverage) find trade-offs between more involved tuples and less noise in join result. Thus, on TPC-H, Employees, and Yelp datasets, all preferences but h_1 perform very well with optimal or near-optimal results. h_1 does not perform very well, as there are often multiple candidate predicates where all the tuples in both tables are involved. However, for IMDb dataset, each table contains both a primary-key ID column and several other foreign-key ID columns, all in the same domain. Thus, many primary/foreign-key characteristics are no longer discriminative. This leads to the bad performances of all preferences, especially h_2, h_3, and h_4. However, h_1 and h_5 performs relatively good, as they rely on coverage which is less affected.

Table III also compares our algorithm and the four baselines (including two versions of "SMaSh"). Our algorithm and the baselines have a distinct difference. We measure the quality of join result directly, while the baselines investigates the characteristics of attributes which affect the join result indirectly. This leads to significant improvement on our accuracy. As we can see, h_3 in our method outperforms all the baselines substantially on all datasets except for IMDb, and h_5 outperforms all the baselines clearly on IMDb. The closest competitor is FastFK, as it only searches for primary/foreign-keys which coincidently define the ground-truths in our experiment setting. However, it is still weaker than ours. The limitation of only returning size-1 predicate also affects FastFK's performance. The second closest competitor is the latest SMaSh, thanks to its set similarities among attribute pairs and post-filtering by strength and coverage. It has the same accuracy of our h_5 on Employees and the same of h_3 on IMDb. The next closest competitor is the AttrCluster algorithm. The algorithm from MIMatch gives the worst results, as it is designed for schema mapping instead of joining hence always matches too many columns. On the Yelp dataset, its results are unavailable due to unacceptable running time (more than 12 hours). Again on the challenging dataset IMDb, all baselines perform poorly.

B. Efficiency and Scalability

Table IV compares our algorithm with the four baselines. Note that SMaSh has the same running time with or without

TABLE IV: Average running times (in sec).

Dataset	Ours					MIMatch	AttrCluster	SMaSh	FastFK
	h_1	h_2	h_3	h_4	h_5				
TPC-H	35.1	62.2	61.8	90.4	134.3	176.7	142.9	19.6	**17.1**
Employees	3.9	5.2	5.3	5.9	9.6	6.1	8.4	**2**	**2**
IMDb	90.2	75.3	75.2	80.8	238.2	344.7	442.1	66.1	**57.3**
Yelp	15.3	16.4	23	290.8	689	n/a	958.4	21.9	**10.6**

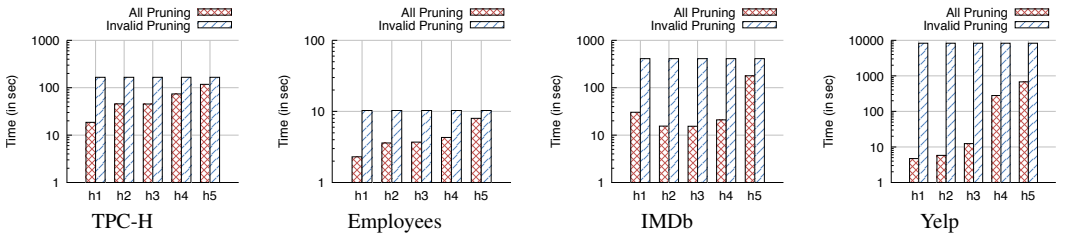

Fig. 5: Average computing times (with and without upper-bound/non-closed predicate pruning).

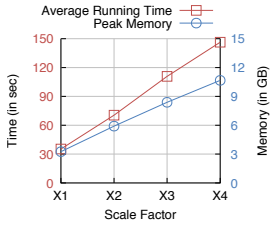

Fig. 6: Scalability of h_1 on TPC-H.

post-filtering. Note that the running time also includes the data loading time. For all algorithms except SMaSh and FastFK, the loading takes significantly long time, especially on large datasets, because of a pre-processing step of mapping each table cell's string value to a unique integer value for later faster value comparison. SMaSh and FastFK are very fast on all the datasets, as they only need to measure each possible attribute pair individually and do not require the mapping step during loading. MIMatch is very slow due to the lack of pruning strategy, and cannot complete on the Yelp dataset. AttrCluster takes more time, mainly for its linear programming solver which is significantly slow.

We evaluate the effectiveness of our pruning techniques. Figure 5 reports the results on using all pruning techniques versus pruning only invalid predicates (discussed at the beginning of Section V-B). Our pruning techniques take clear effect. To remove the disturbance of data loading time, the results are in computing time only.

We conduct a scalability test using the TPC-H dataset which naturally supports setting different scale factors. A scale factor of $\times n$ generates a database with size of nGB. Figure 6 shows the average runtime and peak memory usage using h_1 as the preference, suggesting that our algorithm is highly scalable.

VII. Conclusions

In this paper, we tackled the problem of joining tables automatically to meet user preference on results. We proposed a set of preferences that may be used in various scenarios. We further developed a general algorithm framework for any preference and efficient pruning and speedup techniques. We used 4 datasets to evaluate our method and compare with 4 state of the art baselines. The experimental results demonstrated both the effectiveness of the preferences and the efficiency of our algorithm.

References

[1] TPC-H Benchmark. http://www.tpc.org/tpch.
[2] MySQL Employee Database. http://dev.mysql.com/doc/employee/en.
[3] IMDb Alternative Interfaces. http://www.imdb.com/interfaces.
[4] Yelp Dataset Challenge. http://www.yelp.ca/dataset_challenge.
[5] A. C. Acar and A. Motro. Efficient discovery of join plans in schemaless data. In *IDEAS*'2009.
[6] B. Ahmadi, M. Hadjieleftheriou, T. Seidl, D. Srivastava, and S. Venkatasubramanian. Type-based categorization of relational attributes. In *EDBT*'2009.
[7] A. Bilke and F. Naumann. Schema matching using duplicates. In *ICDE*'2005.
[8] L. Caruccio, G. Polese, and G. Tortora. Synchronization of Queries and Views Upon Schema Evolutions: A Survey. *ACM TODS*, 41(2), 2016.
[9] Z. Chen, V. R. Narasayya, and S. Chaudhuri. Fast Foreign-Key Detection in Microsoft SQL Server PowerPivot for Excel. *PVLDB*, 7(13), 2014.
[10] C. Gao, J. Wang, J. Pei, R. Li, and Y. Chang. Preference-driven Similarity Join. In *WI*'2017.
[11] O. Hassanzadeh, K. Q. Pu, S. H. Yeganeh, R. J. Miller, L. Popa, M. A. Hernndez, and H. Ho. Discovering Linkage Points over Web Data. *PVLDB*, 6(6), 2013.
[12] J. Kang and J. F. Naughton. On schema matching with opaque column names and data values. In *SIGMOD*'2003.
[13] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4), 2001.
[14] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *WebDB*'2009.
[15] R. Rymon. Search through systematic set enumeration. In *KR*'1992.
[16] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. GORDIAN: efficient and scalable discovery of composite keys. In *VLDB*'2006.
[17] K. Stefanidis, M. Drosou, and E. Pitoura. PerK: personalized keyword search in relational databases through preferences. In *EDBT*'2010.
[18] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM TODS*, 36(3), 2011.
[19] R. H. Warren and F. W. Tompa. Multi-column Substring Matching for Database Schema Translation. In *VLDB*'2006.
[20] X. Yuan, X. Cai, M. Yu, C. Wang, Y. Zhang, and Y. Wen. Efficient Foreign Key Discovery Based on Nearest Neighbor Search. In *WAIM*'2015.
[21] C. J. Zhang, L. Chen, H. V. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 6(9), 2013.
[22] C. J. Zhang, Z. Zhao, L. Chen, H. V. Jagadish, and C. C. Cao:. Crowdmatcher: crowd-assisted schema matching. In *SIGMOD*'2014.
[23] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *PVLDB*, 3(1), 2010.
[24] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. Automatic discovery of attributes in relational databases. In *SIGMOD*'2011.