Data Integration on Complex Data

by

Chuancong Gao

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Science

© Chuancong Gao 2019
SIMON FRASER UNIVERSITY
Summer 2019

Approval

Name:	**Chuancong Gao**
Degree:	**Master of Science**
Title:	**Data Integration on Complex Data**
Examining Committee:	**Chair:** Peter Chow-White Professor
	Jian Pei Senior Supervisor Professor
	Ke Wang Supervisor Professor
	Jiannan Wang Internal Examiner Assistant Professor
Date Defended:	**August 22nd, 2019**

Abstract

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. 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. In this thesis, we first tackle this data integration challenge by investigating how to join tables guided by user preferences.

To further improve the quality of data integration, data value similarity needs to be considered as well. Threshold-driven similarity join has been extensively studied in the past. However, the process of tuning similarity threshold is tedious and error-prone. In this thesis, when performing a similarity join, we further seek to provide a few user preferences instead of similarity thresholds for a user to select from. Once a particular preference is chosen, we automatically tune the threshold and return the corresponding similarity join result.

Comparing to state-of-the-art baselines, our work provide significantly better effectiveness while having comparable efficiency and scalability.

Keywords: Schemaless Join, Similarity Join, User Preference

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the big data era, data often comes from different sources, and the value of data can only be fully extracted by integrating various sources together. Thus, the study of how to integrate data across multiple sources has rapidly grown into one of the most attractive research hotspots.

In this thesis, we focus on how to conduct fully unsupervised data integration both effectively and efficiently. In general, our approaches are both driven by the concept of result set preference, to efficiently/effectively compute and rank the possible data integration results based user-provided preference at running time. Each preference is modeled based on user requirement from certain aspect, thus able to drive near-optimal results for many cases when choosing properly. As data integration is done in unsupervised manner by data-independent preference at the macro/requirement level instead of data-specific parameters at the micro/engineering level, the significant burden of parameter tuning by user is mostly avoided. Overall, we believe our work further extend the boundaries of current research and push the limits of performance.

1.1 Motivation

We motivate our work by introducing a real-world problem of data integration across multiple sources as follows.

Example 1 (Data integration for NCAA sports across multiple sources)**.** *Let's consider the famous National Collegiate Athletic Association (NCAA) sports as an example. It covers multiple sports, including baseball, basketball, football, hockey, etc. For example, in game season 2015 for men's basketball alone, there are around* $1,100$ *teams and* $32,000$ *games across three divisions. Due to its vast popularity, not only professionals but also viewers closely follow the statistics and analysis of each game.*

However, for each game, the data does not come from a single source. In fact, there are at least three official data sources.

- *Even NCAA itself provides two different versions of standard data on its two websites* `www.ncaa.com` *and* `stats.ncaa.org`*. Unless explicitly stated, in this thesis, we refer NCAA data to its latter version on* `stats.ncaa.org`*.*

- *Each university has its own website featuring game data. For example, Simon Fraser University (SFU) provides its data on* `athletics.sfu.ca`*. Besides the standard data like NCAA, universities like SFU also cover additional data like player biography and exhibition games. However, the opponent information is usually not well covered.*

- *Furthermore, each university usually belongs to at least one sport conference. For example, SFU belongs to Great Northwest Athletic Conference (GNAC), which contains* 11 *members. Conferences like GNAC usually provide more detailed data than NCAA, while more overall data than universities, on their own websites like* `www.gnacsports.com`*, etc.*

There are usually multiple data tables for each game, containing statistics covering different aspects of the game, like box score, play-by-play, roster, score information, facility information, etc. For example, there are usually at least 10 *to* 15 *data tables for each men's basketball game on NCAA website. However, as shown in Figure 1.1, even for the most basic box score table, the three sources provide quite different sets of attributes with different headers and footers.*

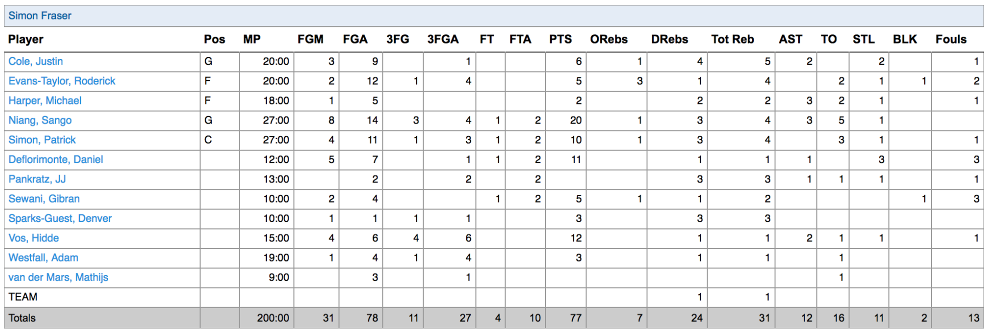
Naturally, people hope to gain access to a knowledge-base for all the NCAA games, which covers all the data across difference sources. While there are attempts in industry, they rely on manually crafted rules to link and merge the data. Thus, both the flexibility and the scalability are significantly limited. □

When building a knowledge-base across multiple data sources, there are usually multiple encountered scenarios that require concrete technical solutions. We cover two of the most important and popular ones in this thesis as follows.

Scenario 1 (Merging multiple relational tables without primary/foreign-key)**.** *The first scenario we encounter is merging multiple relational tables into a larger relational table. Note that the relational tables can either refer to different types of data, like player data and team data within NCAA, or refer to different versions of the same type of data, like player data across different sources as shown in Figure 1.1.*

When primary/foreign-keys exist, this scenario can be solved by finding and joining the primary-keys and their respective foreign-keys. However, for web data or data across different sources, primary/foreign-keys are usually not explicitly defined thus do not exist. Therefore, data can only be merged by determining the proper pairs of columns to be joined across tables according to column names and/or shared data values. □

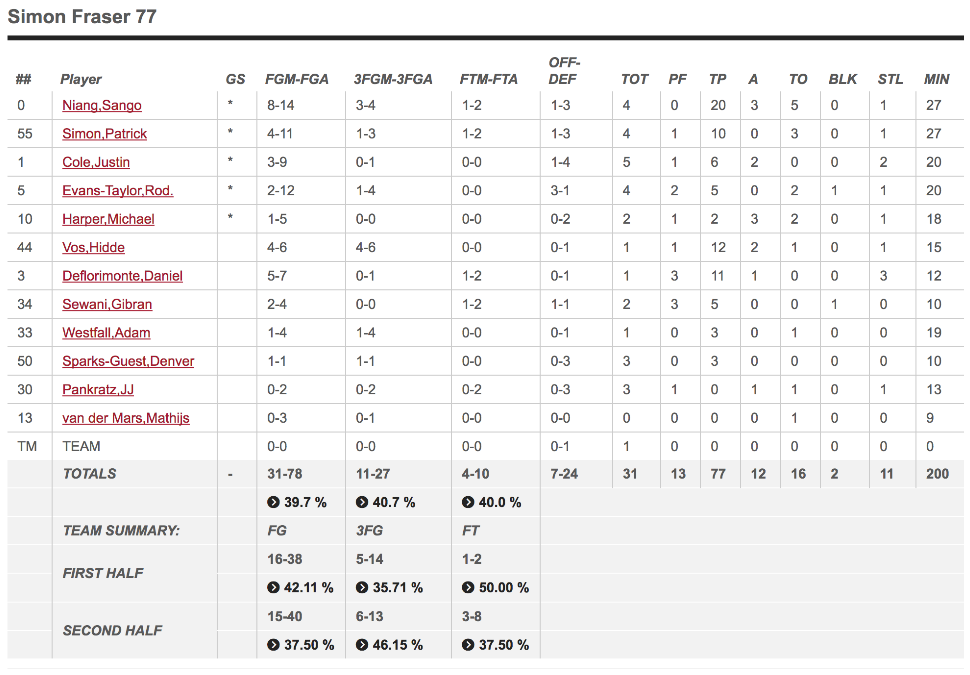
Scenario 2 (Matching similar objects without explicit similarity threshold)**.** *Occasionally, we are able to hold the schema of data, thus knowing the pairs of columns to be joined.*



Simon Fraser

Player	Pos	MP	FGM	FGA	3FG	3FGA	FT	FTA	PTS	ORebs	DRebs	Tot Reb	AST	TO	STL	BLK	Fouls
Cole, Justin	G	20:00	3	9		1			6	1	4	5	2		2		1
Evans-Taylor, Roderick	F	20:00	2	12	1	4			5	3	1	4		2	1	1	2
Harper, Michael	F	18:00	1	5					2		2	2	3	2	1		1
Niang, Sango	G	27:00	8	14	3	4	1	2	20	1	3	4	3	5	1		
Simon, Patrick	C	27:00	4	11	1	3	1	2	10	1	3	4		3	1		1
Deflorimonte, Daniel		12:00	5	7		1	1	2	11		1	1	1		3		3
Pankratz, JJ		13:00		2		2		2			3	3	1	1	1		1
Sewani, Gibran		10:00	2	4			1	2	5	1	1	2				1	3
Sparks-Guest, Denver		10:00	1	1	1	1			3		3	3					
Vos, Hidde		15:00	4	6	4	6			12		1	1	2	1	1		1
Westfall, Adam		19:00	1	4	1	4			3		1	1		1			
van der Mars, Mathijs		9:00		3		1								1			
TEAM											1	1					
Totals		200:00	31	78	11	27	4	10	77	7	24	31	12	16	11	2	13

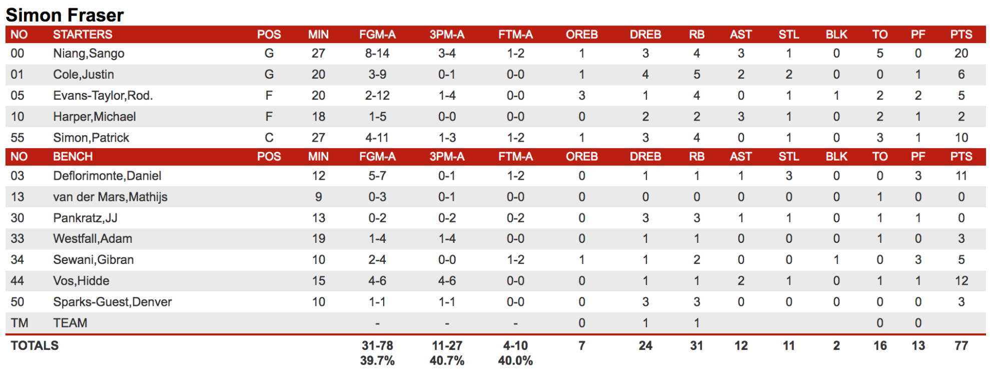
(a) Box score data from NCAA.



Simon Fraser 77

##	Player	GS	FGM-FGA	3FGM-3FGA	FTM-FTA	OFF-DEF	TOT	PF	TP	A	TO	BLK	STL	MIN
0	Niang,Sango	*	8-14	3-4	1-2	1-3	4	0	20	3	5	0	1	27
55	Simon,Patrick	*	4-11	1-3	1-2	1-3	4	1	10	0	3	0	1	27
1	Cole,Justin	*	3-9	0-1	0-0	1-4	5	1	6	2	0	0	2	20
5	Evans-Taylor,Rod.	*	2-12	1-4	0-0	3-1	4	2	5	0	2	1	1	20
10	Harper,Michael	*	1-5	0-0	0-0	0-2	2	1	2	3	2	0	1	18
44	Vos,Hidde		4-6	4-6	0-0	0-1	1	1	12	2	1	0	1	15
3	Deflorimonte,Daniel		5-7	0-1	1-2	0-1	1	3	11	1	0	0	3	12
34	Sewani,Gibran		2-4	0-0	1-2	1-1	2	3	5	0	0	1	0	10
33	Westfall,Adam		1-4	1-4	0-0	0-1	1	0	3	0	1	0	0	19
50	Sparks-Guest,Denver		1-1	1-1	0-0	0-3	3	0	3	0	0	0	0	10
30	Pankratz,JJ		0-2	0-2	0-2	0-3	3	1	0	1	1	0	1	13
13	van der Mars,Mathijs		0-3	0-1	0-0	0-0	0	0	0	0	1	0	0	9
TM	TEAM		0-0	0-0	0-0	0-1	1	0	0	0	0	0	0	0
	TOTALS	-	**31-78**	**11-27**	**4-10**	**7-24**	**31**	**13**	**77**	**12**	**16**	**2**	**11**	**200**
			39.7 %	**40.7 %**	**40.0 %**									
	TEAM SUMMARY:		**FG**	**3FG**	**FT**									
	FIRST HALF		**16-38**	**5-14**	**1-2**									
			42.11 %	**35.71 %**	**50.00 %**									
	SECOND HALF		**15-40**	**6-13**	**3-8**									
			37.50 %	**46.15 %**	**37.50 %**									

(b) Box score data from SFU.



Simon Fraser

NO	STARTERS	POS	MIN	FGM-A	3PM-A	FTM-A	OREB	DREB	RB	AST	STL	BLK	TO	PF	PTS
00	Niang,Sango	G	27	8-14	3-4	1-2	1	3	4	3	1	0	5	0	20
01	Cole,Justin	G	20	3-9	0-1	0-0	1	4	5	2	2	0	0	1	6
05	Evans-Taylor,Rod.	F	20	2-12	1-4	0-0	3	1	4	0	1	1	2	2	5
10	Harper,Michael	F	18	1-5	0-0	0-0	0	2	2	3	1	0	2	1	2
55	Simon,Patrick	C	27	4-11	1-3	1-2	1	3	4	0	1	0	3	1	10
NO	BENCH	POS	MIN	FGM-A	3PM-A	FTM-A	OREB	DREB	RB	AST	STL	BLK	TO	PF	PTS
03	Deflorimonte,Daniel		12	5-7	0-1	1-2	0	1	1	1	3	0	0	3	11
13	van der Mars,Mathijs		9	0-3	0-1	0-0	0	0	0	0	0	0	1	0	0
30	Pankratz,JJ		13	0-2	0-2	0-2	0	3	3	1	1	0	1	1	0
33	Westfall,Adam		19	1-4	1-4	0-0	0	1	1	0	0	0	1	0	3
34	Sewani,Gibran		10	2-4	0-0	1-2	1	1	2	0	0	1	0	3	5
44	Vos,Hidde		15	4-6	4-6	0-0	0	1	1	2	1	0	1	1	12
50	Sparks-Guest,Denver		10	1-1	1-1	0-0	0	3	3	0	0	0	0	0	3
TM	TEAM			-	-	-	0	1	1				0	0	
TOTALS				31-78 39.7%	11-27 40.7%	4-10 40.0%	7	24	31	12	11	2	16	13	77

(c) Box score data from GNAC.

Figure 1.1: Box score data of the same men's basketball game from various sources.

However, even with full knowledge of schema, straight-forward equi-join may still fail, due to the possibility that the data values do not fully match. For example, for the basketball game in Figure 1.1, player name "Evans-Taylor, Roderick" sometimes appears as "Evans-Taylor, Rod" or even "Taylor-Evans, R".

Instead of equi-join, similarity-join can be used to link two sets of similar values, by finding the pairs of similar values whose similarities are above a specified threshold. However, selecting an appropriate threshold may be far from easy. It is extremely difficult for humans to figure out the effect of different thresholds on the quality of joined result, as choosing a good threshold depends on the domain knowledge of not only the specific task but also the underlying data. □

1.2 Challenges

For each of the two scenarios, there are multiple challenges to be considered and conquered.

Challenge 1 (How to achieve high quality by unsupervised approach)**.** *To achieve good quality of data integration, some supervised methods manually label part of data as the ground truth of integration. However, human labeling lacks both timeliness and scalability significantly, due to its nature of time-consuming and expensiveness, respectively. For example, for each NCAA game, the post-game analysis and statistics are usually required by both the team and the viewers right after the finish of game. Given the huge daily amount of NCAA games (hundreds during game season), this task cannot be easily done by human.*

Some other supervised methods utilize external knowledge as ground truth. However, the coverage of external knowledge is always limited to certain scope. Thus, long-tail or fresh information may not able to be covered at all. For example, for each NCAA game, a knowledge-base needs to be consistently synchronized to reflect changes of players and teams in each season.

The limitations of supervised approach raise the following questions. How to achieve high quality of data integration by unsupervised approach? In particular, we consider the following aspects. First, how to measure the quality of data integration for each scenario? Second, how to design the respective unsupervised mechanism of data integration for each scenario? □

Challenge 2 (How to achieve high efficiency and scalability)**.** *Besides the quality of data integration, the efficiency and scalability are also essential. For the first scenario, there are exponential number of sets of column pairs to join even two relational tables. Even worse, for each relational table, there may be millions of records. For the second scenario, there are at most n^2 number of similarity thresholds to perform similarity join between two sets of n objects, where n could be millions.*

The above facts of each scenario raise the following questions. First, how do we speed up the computation for each candidate integration result w.r.t. each scenario? Second, can *we reduce the number of candidates without jeopardizing the quality of integration? Third, is there any shared computation between two candidates to speed up?* □

Challenge 3 (How to achieve high adaptability and extendability)**.** *While we use the example of NCAA sports in Example 1 to help motivate the scenarios, the methods are expected to be highly adaptable and extendable, therefore enable the possibilities of applying to other problem settings.*

This requirement raises the following questions. For each scenario, can the same method be applied to different problem settings? Specifically, for the second scenario, can the same method be further applied to different types of similarity functions? □

1.3 Contributions

In this thesis, we study the above two scenarios in Section 1.1 and address the three challenges in Section 1.2. We establish the following contributions.

Contribution 1 (Result set preferences for quality)**.** *To address the first challenge w.r.t. the quality of unsupervised approach, we carefully devise several result set preferences for each scenario.*

Our key insight is inspired by the concept of preference in areas like economics, which is an ordering of different alternatives (results) [3]. Taking Yelp.com as an example, the restaurants can be ranked in different ways, such as by distance, price, or rating. The different ordering may meet different search intents. A user needs to evaluate her query intent and choose the most suitable ranking accordingly.

Similarly, under different preferences there are different ways to integrate the data. From a user's point of view, she only needs to understand and specify the preference over the potential integration results at the macro-level, and the best integration result w.r.t. the preference are computed automatically without user interaction at the micro-level. To the best of our knowledge, this research direction has not been touched in literature.

Specifically, for each scenario, we devise the respective result set preferences accordingly as follows. For schemaless join, we propose a series of 5 *preferences according to different user intents, like maximizing the number of joining groups, minimizing the size of outer join, etc. Similarly, for similarity join, we propose two preferences, including maximizing the number of joining groups and minimizing the size of outer join.* □

Contribution 2 (Advanced algorithms for efficiency and scalability)**.** *To address the second challenge w.r.t. efficiency and scalability, for each scenario, we devise the respective algorithm accordingly as follows to speed up. For schemaless join, we propose an efficient enumeration framework for join predicate, and further develop multiple pruning techniques by investigating the monotonicity of each preference. For similarity join, we propose a novel* *similarity join framework with preference, along with multiple effective optimization techniques by investigating the monotonicity of each preference.* □

Contribution 3 (Flexible frameworks for adaptability and extendability)**.** *To address the third challenge w.r.t. adaptability and extendability, for each scenario, we devise the respective algorithm framework accordingly as follows. For schemaless join, we propose a universal framework which works for any result set preference with monotonicity. For similarity join, we propose a universal framework for any preference with monotonicity. We further extend the framework for handling all the common set-based string similarities.* □

1.4 Organization

We structure the remainder of this thesis as follows.

- We present the related work w.r.t. each scenario in detail in Chapter 2, covering schemaless join and similarity join.
- Chapter 3 presents our work about schemaless join for result set preference. We introduce a series of result set preferences along with their respective properties of monotonicity, and propose the respective algorithm framework for the enumeration of candidate integration results.
- Chapter 4 presents our work about preference-based similarity join. We introduce two result set preferences along with their respective properties of monotonicity, and propose the respective algorithm framework for similarity join with preference.
- Finally, in Chapter 5, we summarize the proposed methods and conclude the thesis. We further discuss some interesting and promising future work.

Chapter 2

Related Work

By combining heterogeneous data from differences sources, data integration offers users a result that is not only unified but also meaningful. As a complex data processing concept, data integration involves different tasks, including data cleaning, ETL (extract, transform, and load), data modeling, data profiling, data migration, etc. However, at its core are data mapping tasks that discover the relationships among different data sources. In this chapter, we first study four different data mapping tasks: schema matching, attribute augmentation, foreign-key discovery, and attribute clustering, then we further study data mapping over data value similarity.

2.1 Schemaless Join

In this section we introduce the 4 core tasks of data integration for data mapping between different sources in detail.

2.1.1 Schema Matching

Given multiple relations where each relation has its own single-relation schema but no cross-relation schema, schema matching tries to derive mapping rules of attributes among the relations. For each mapping, an attribute of a relation can be mapped to or from different numbers of attributes of different relations. Thus, different mappings have different cardinalities, namely $1 : 1$, $1 : n$, $n : 1$, and $n : m$.

Kang and Naughton [18] 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 [8] found mapping between attributes in different tables. Warren and Tompa [28] conducted schema mappings by finding string transformation rules between attributes. Acar and Motro [1] 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.* [44] 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.

Two famous previous studies [46, 60] summarize most of the schema matching works before 2001 and 2011, respectively. However, there are several new methods proposed after 2011, such as [44, 34, 35, 61]. Instead of improving the very matured schema-level methods, most of the new methods either study the instance-level methods or hybrid of schema-level and instance-level methods, or utilize new kinds of external knowledge like web tables or human knowledge.

2.1.2 Attribute Augmentation

Assume a query table and a set of reference tables, the attribute augmentation task augments the missing values of the query table derived from the reference tables, by trying to establish mappings and joining each reference table with query table. A primary-key of query table may also be specified in advance to facilitate the process.

Various works have studied this problem since 2009, like [62, 63, 64, 65]. They consider different forms of the query table. Most of them require the existence of a primary-key in the query table, while [63] only requires the column name of each column. Given the vast amount and high availability, web tables are often used as the references tables in different ways, including real time [62, 64] vs. offline [63, 65] table crawling, with [63] or without [62, 64, 65] surrounding context, etc.

2.1.3 Foreign-Key Discovery

Given the primary-key of each relation, the foreign-key discovery task tries to find the unknown foreign-keys among different relations to join with the primary-key. This is not a trivial problem. As we know, within one relation, there may be many candidate foreign-keys satisfying the primary/foreign-keys constraint. Even worse, some of the primary/foreign-key may contain multiple columns. Given the large number of candidate foreign-keys, how to efficiently enumerate them and how to effectively rank them are both challenging.

Adopting machine learning techniques, Rostin *et al.* [47] 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.* [36] put a primary-key column and its corresponding foreign-key columns into the same cluster, and found all foreign-key columns. [36] is also capable of finding multi-column foreign-keys. Recently, [43] 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. [33] 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.

Various methods [47, 36, 43, 33] have been proposed for this task. Each of them has its own way of enumerating and ranking candidate foreign-keys. However, they all have different limitations. [36] is the only one capable of finding multi-column foreign-keys. While most of the works for foreign-key discovery also assume knowing the primary-key of each relation, only [43] does not assume any prior knowledge about the primary-key. In practice, the assumption is sometimes difficult to achieve. Although there are some works like [49], which finds all the candidate-keys by checking whether the values in the key attributes are unique, the primary-key is still difficult to identify.

2.1.4 Attribute Clustering

As its name indicates, given a set of relations, the attribute clustering task tries to assign the columns from different relations into disjoint clusters, where the columns in each cluster are equivalent semantically. If two attributes have one of the below relationships, they are considered as semantically equivalent, as defined in [37]:

- Primary/foreign-key;
- Two foreign-keys referring to the same primary-key;
- A column in a view and the corresponding column in the base table;
- Two columns in two views but from the same column in the base table;
- No explicit relationship but share the same semantic meaning (e.g., customer name columns from different tables).

While the first four relationships are easy to be identified, inferring the last relationship is very challenging [37].

Ahmadi *et al.* [2] 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.* [37] 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 [36]. In [37], 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.

[2, 37] are two known methods proposed for this task. The former one treats all the values as strings, and extracts q-grams as features. The latter one clusters the columns into distribution clusters and further clustered into attribute clusters, by computing EMD (earth mover distance) and further intersection EMD between each column pair, respectively.

2.2 Similarity Join

Due to the crucial role of similarity join in data integration and data cleaning, numerous similarity join algorithms have been proposed [5, 12, 7, 32, 30, 31, 25, 55, 26, 17]. There are also scalable implementations of the algorithms using the MapReduce framework [24, 56, 13]. Top-k similarity join is also explored [31, 19].

While the majority of the existing work on similarity join needs to specify a similarity threshold or a limit of the number of results returned, there do exist some studies that seek to find a suitable threshold for similarity join in a supervised way [27, 11, 6, 10]. Both [6] and [10] adopted active learning to tweak the threshold. Chaudhuri *et al.* [11] learned an operator tree, where each node contains a similarity threshold and a similarity function on each of the splitting attributes. Wang *et al.* [27] modeled this problem as an optimization problem and applied hill climbing to optimize the threshold-selection process. To the best of our knowledge, [16] is the first work to discuss the one-to-one, one-to-many, and many-to-many constraints for similarity join.

Chapter 3

Schemaless Join for Result Set Preferences

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 [46, 18, 8, 28, 44] and attribute clustering [2, 37, 36, 37]. 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.

In this chapter, 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 [34, 35]. 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 3.1. 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 3.1 we add some column names so that our discussion is easy to follow. Such column names are unavailable to our algorithms.

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)

Figure 3.1: A toy example of two tables.

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 3.2, 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 3.2, 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 chapter, 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 chapter is organized as follows. We present our problem formulation in Section 3.1, discuss the preferences in Section 3.2, and develop our algorithm in Section 3.3. We report an empirical study in Section 3.4.

3.1 Problem Definition

Consider two tables R and S. Denote by $\mathcal{C}^R = \left\{c_1^R, c_2^R, \ldots, c_m^R\right\}$ and $\mathcal{C}^S = \left\{c_1^S, c_2^S, \ldots, c_n^S\right\}$ 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 chapter. 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 3.1. 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 3.2 gives a few examples of inner join using various predicates on the tables in Figure 3.1.

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

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

Figure 3.2: Examples of $R \bowtie_{p_i} S$ using each predicate p_i.

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

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

For example, in Figure 3.2, 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 $(null, \dots)_n$, and another two additional tuples, resulted from joining $(null, \dots)_m$ 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 $p \subset q$, $R \bowtie_p S \supseteq R \bowtie_q S$.* $\square$

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 $R \bowtie_p S \subseteq R \bowtie_{p \cup q} S$. $\square$

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

For example, for the two tables in Figure 3.1, 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 3.2. 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 $R \bowtie_p S = R \bowtie_q S$.* $\square$

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, $K^{R,S}(p) = \pi_{c^R_{i_1}, c^R_{i_2}, \ldots, c^R_{i_{|p|}}}(R \bowtie_p S)$, where $p = \left\{ \left(c^R_{i_1}, c^S_{j_1}\right), \left(c^R_{i_2}, c^S_{j_2}\right), \ldots, \left(c^R_{i_{|p|}}, c^S_{j_{|p|}}\right) \right\}$. When R and S are clear from context, we just write $K(p)$ instead of $K^{R,S}(p)$. For each joining key $k = \left(v_1, v_2, \ldots, v_{|p|}\right) \in K(p)$, we define the corresponding **joining group** as a pair $\left(G^R_k, G^S_k\right)$, where G^R_k and G^S_k are the sets of tuples in R and those in S, respectively, having the same key, that is, we have $G^R_k = \sigma_{c^R_{i_1}=v_1 \wedge c^R_{i_2}=v_2 \wedge \ldots \wedge c^R_{i_{|p|}}=v_{|p|}}(R)$ and $G^S_k = \sigma_{c^S_{j_1}=v_1 \wedge c^S_{j_2}=v_2 \wedge \ldots \wedge c^S_{j_{|p|}}=v_{|p|}}(S)$. Moreover, we denote by $r_k = \left|G^R_k\right|$ and $s_k = \left|G^S_k\right|$.

Take $p_2 = \{(\text{Tel.}, \text{Tel.}), (\text{Name}, \text{Client}), (\text{Home}, \text{Address})\}$ in Figure 3.2 as an example. $R \bowtie_{p_2} S$ 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 $\left(G^R_{k_1}, G^S_{k_1}\right)$, where $G^R_{k_1}$ is a set of tuples of ID P1 and $G^S_{k_1}$ 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 3.2. 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 $h_0(p) = |R \bowtie_p S|$. Obviously, for any predicates p and p' such that $p \subseteq p'$, $R \bowtie_p S \supseteq R \bowtie_{p'} S$ 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 $h(p)$.* □

3.2 Preferences on Join Results

In this section we formulate a series of preferences on join results that capture different needs in various applications. Table 3.1 summarizes the preferences. Those preferences cover the most common scenarios.

3.2.1 Preference for More Tuples Joined

In our example in Figure 3.1, 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

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

[44] 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.

3.2.2 Preference for More Joining Groups

In our example in Figure 3.1, 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 [47]. Formally, we define preference scoring function

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

3.2.3 Preference for Less Tuples in Outer Join

In our example in Figure 3.1, 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 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

Table 3.1: Summary of the preferences.

Preference Description	Scoring Function	Monotonic Upper-Bound Function
Maximizing #involved tuples (coverage)	$h_1(p) = \sum_{k \in K(p)} (r_k + s_k)$	$h_1(p)$
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 & strength	$h_5(p) = \frac{2 \cdot \sum_{k \in K(p)} (r_k + s_k)}{2 \cdot \lvert R \bowtie_p S \rvert + \lvert R \rvert + \lvert S \rvert}$	$\widehat{h_5}(p) = \frac{2 \cdot \sum_{k \in K(p)} (r_k + s_k)}{\sum_{k \in K(p)} (r_k + s_k) + \lvert R \rvert + \lvert S \rvert}$

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 $|R \bowtie_p S| = \sum_{k \in K(p)} r_k \cdot s_k$ and further $\arg\min_p \ (|R| + |S| - h_3(p)) = \arg\min_p \ \left| R \bowtie_p^{\leftrightarrow} S \right| = \arg\max_p \ h_3(p)$.

3.2.4 Preference for More Balanced Distribution of Joining Keys

Consider our example in Figure 3.1, 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. [36] 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

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

3.2.5 Preference for Greater Mean of Coverage and Strength

Consider a task of the example in Figure 3.1. 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 [44] for better schema matching accuracy, and defined as $str(p) = \frac{\sum_{k \in K(p)} (r_k + s_k)}{2 \cdot |R \bowtie_p S|}$. Usually, the more attributes joined, the higher confidence we have on the matched records in the join result. 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

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 $\exists p' <^{lex} p : p' \supset p \wedge |R \bowtie_{p'} S| = |R \bowtie_p S|$ 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 $(c^R_{l_i}, c^S_{w_j})$ by multi-way merge join on $L^R_{l_i}$ and $L^S_{w_j}$ for all the
    joining groups of $p$ do
6      $enumerate(p \cup (c^R_{l_i}, c^S_{w_j}))$              // recursively extend $p$
```

strength as a preference scoring function.

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

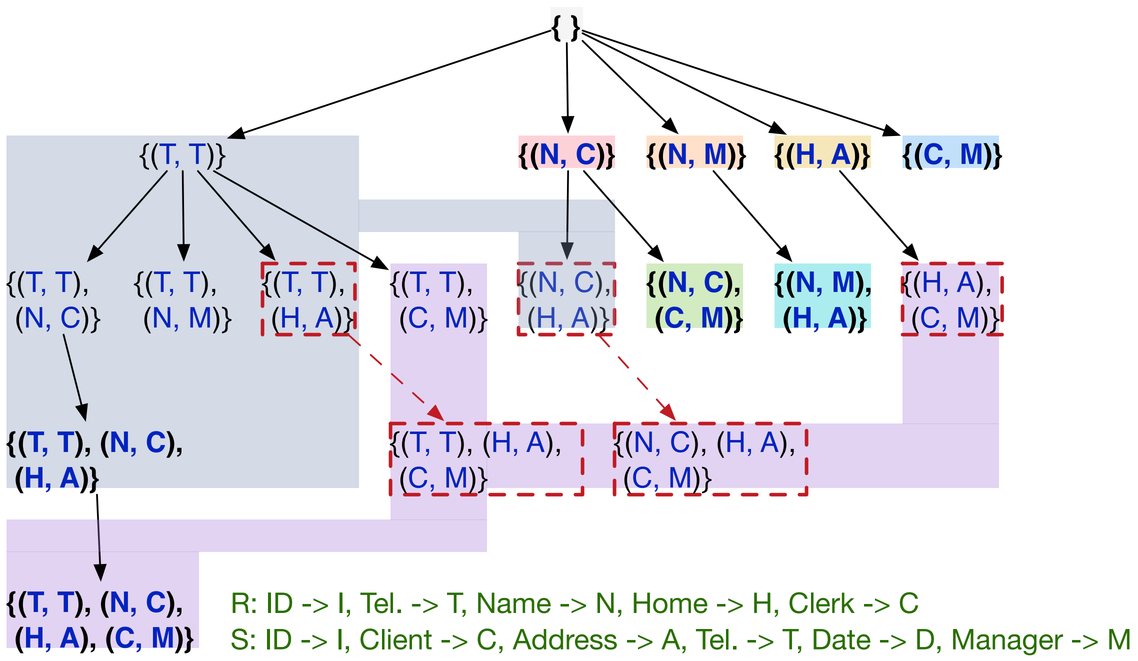
3.3 Finding Best Predicates

Given a preference on result sets defined in Section 3.2, 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 3.2.

3.3.1 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 [48]. 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 a is ordered before b. For example, in Figure 3.1, 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



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.

Figure 3.3: The predicate enumeration tree of valid predicates.

(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 3.1 $\{(\text{Tel.}, \text{Tel.})\} <^{lex} \{(\text{Tel.}, \text{Tel.}), (\text{Name}, \text{Client})\}$. 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 3.3 shows the set enumeration tree in our example in Figure 3.1.

As established by the set enumeration tree construction [48], 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.

3.3.2 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.

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.

Non-Closed Predicate Pruning

To explore how to prune non-closed predicates, let us look at an example. Consider the tables in Figure 3.2 and the predicates in Figure 3.3. We can verify that predicate $p = \{(\text{Name}, \text{Client}), (\text{Home}, \text{Address})\}$ is not closed, as for predicate $p_2 = \{(\text{Tel.}, \text{Tel.})\} \cup p$, $R \bowtie_{p_2} S = R \bowtie_p S$. Therefore, every tuple in $R \bowtie_p S$ 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 $(\text{Tel.}, \text{Tel.})$, like $q = \{(\text{Name}, \text{Client}), (\text{Home}, \text{Address}), (\text{Clerk}, \text{Manager})\}$, we can always add $(\text{Tel.}, \text{Tel.})$ to obtain another predicate $q' \supset q$ where $R \bowtie_q S = R \bowtie_{q'} S$. Thus, q cannot be closed.

We formulate the above observation as follows.

Theorem 3. *If for predicates x and p, $|R \bowtie_p S| = |R \bowtie_{p \cup x} S|$. 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, $R \bowtie_p S \supseteq R \bowtie_q S$. Thus, $R \bowtie_p S = R \bowtie_q S$ if and only if $|R \bowtie_p S| = |R \bowtie_q S|$.

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

For predicate p, we check if there exists any previous enumerated predicate p' where $p' \supset p$ and $|R \bowtie_{p'} S| = |R \bowtie_p S|$. 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 $|R \bowtie_{p'} S| = |R \bowtie_p S|$ 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 3.3, the whole subtree rooted at predicate $\{(\text{Name}, \text{Client}), (\text{Home}, \text{Address})\}$ can be pruned.

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 3.1 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 $a \times b \in R \bowtie_q S$ only if $a \times b \in R \bowtie_p S$. Since $\sum_{k \in K(p)} r_k = |\{a \in R \mid \exists b \in S : a \times b \in R \bowtie_p S\}|$, $\sum_{k \in K(p)} r_k \geq \sum_{k \in K(q)} r_k$. 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. $\square$

h_2 is not monotonic. Consider predicate p_4 in Figure 3.2 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. $\square$

h_3 is not monotonic, either. For example, in Figure 3.2, $p_3 \subset p_2$, 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, $h_3(p) \leq \sum_{k \in K(p)} (r_k + s_k) -$
$\max\left\{\sum_{k \in K(p)} r_k, \sum_{k \in K(p)} s_k\right\} = \min\left\{\sum_{k \in K(p)} r_k, \sum_{k \in K(p)} s_k\right\} = \widehat{h_3}(p)$. $\square$

h_4 is not monotonic. In our example in Figure 3.1, $p_3 \subset p_2$, $h_4(p_3) = 1.447 < h_4(p_2) = 1.5$. 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 3.1, $p_1 \subset p_2$, $h_5(p_1) = 0.692 < h_5(p_2) = 0.7$.

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. $\square$

3.3.3 Producing Children Nodes using Multi-Way Merge Joins

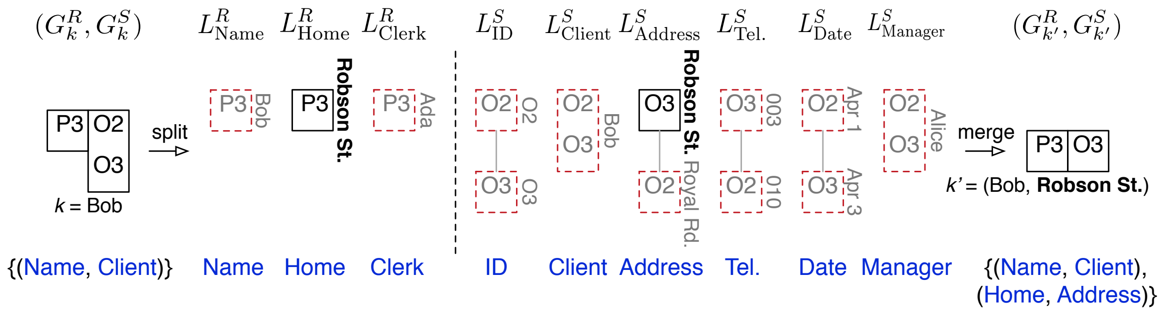
Let $p = \left\{ \left(c_{i_1}^R, c_{j_1}^S\right), \left(c_{i_2}^R, c_{j_2}^S\right), \ldots, \left(c_{i_{|p|}}^R, c_{j_{|p|}}^S\right) \right\}$ be any predicate in the predicate set enumeration tree. A central step in our algorithm is to produce the children nodes of p. A brute-force method can enumerate 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 3.2. The candidate attributes are three attributes "Name", "Home", and "Clerk"



Values in bold are matched. For each tuple, we only show its ID.

Figure 3.4: Example of multi-way merge join for splitting the second joining group of predicate $\{(\text{Name}, \text{Client})\}$.

Table 3.2: 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

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 3.4 gives an example of the second joining group, which has the joining key k = Bob and contains tuples P3 in R and O2 and O3 in S. When splitting the joining group, we get the sorted list L^R_{Home} containing P3 with value "Robson St.", w.r.t. attribute "Home" in R. Similarly, we get the sorted list L^S_{Address} 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^R_{Home} is matched with O3 in L^S_{Address} 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.

Table 3.3: 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

3.4 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 3.2 shows their statistical information.

- The standard benchmark dataset TPC-H [38], which has about 8 million records in total in 8 tables.
- A smaller sample dataset Employees [39] from MySQL, containing about 4 million records in total across 6 tables.
- IMDb [40] [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 [41] 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 "MI-Match" [18] 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.

[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/`).

The second baseline "AttrCluster" [37] 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 $\left\{c_1^R, c_2^R, c_1^S\right\}$ and $\left\{c_3^R, c_2^S\right\}$ are converted to two predicates $\left\{\left(c_1^R, c_1^S\right), \left(c_2^R, c_1^S\right)\right\}$ and $\left\{\left(c_3^R, c_2^S\right)\right\}$, 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" [44] finds a set of matching attribute pairs for schema matching. Each pair is evaluated individually by the algorithm. We choose Jaccard to measure 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 [44], 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" [43] 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.

3.4.1 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-

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

Table 3.4: 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**

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 3.3 compares the 5 preferences on the 4 datasets. As the ground-truths are primary/foreign-keys in our experiments, 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 [47], 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 3.3 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

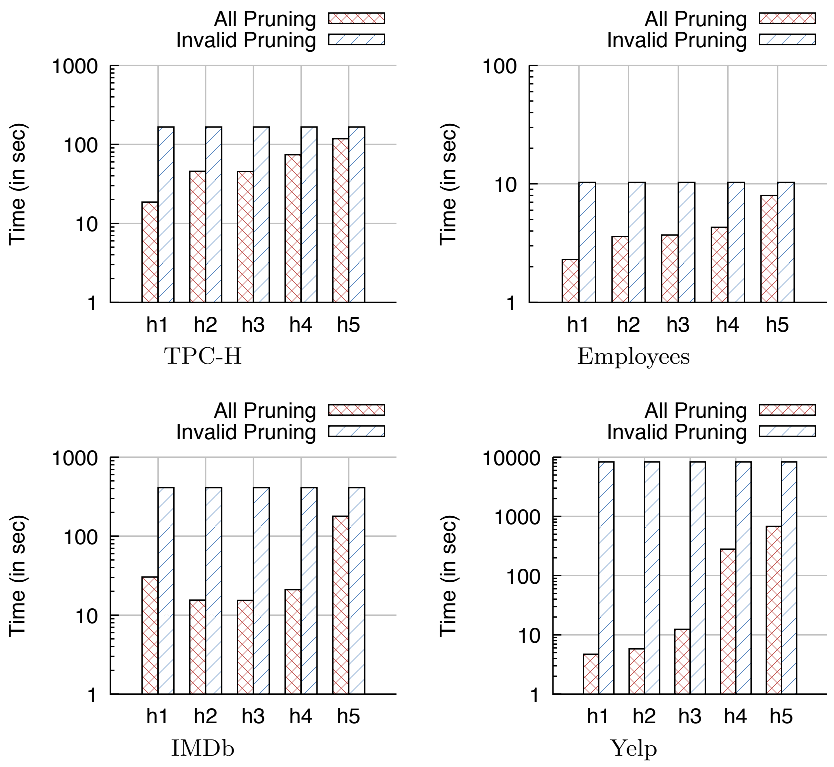


Figure 3.5: Average computing times (with and without upper-bound/non-closed predicate pruning).

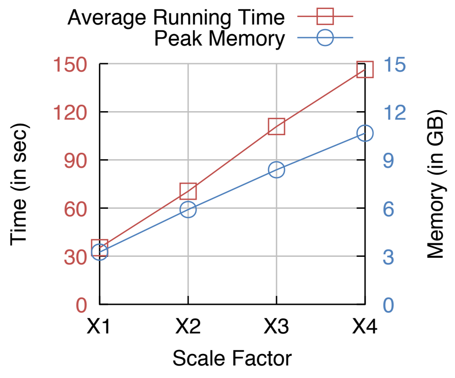


Figure 3.6: Scalability of h_1 on TPC-H.

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.

3.4.2 Efficiency and Scalability

Table 3.4 compares our algorithm with the four baselines. Note that SMaSh has the same running time with or without 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 3.5 reports the results on using all pruning techniques versus pruning only invalid predicates (discussed at the beginning of Section 3.3.2). 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 3.6 shows the average runtime and peak memory usage using h_1 as the preference, suggesting that our algorithm is highly scalable.

Chapter 4

Preference-based Similarity Join

A key characteristic of big data is *variety.* Data (especially web data) often comes from different sources and the value of data can only be extracted by integrating various sources together. Similarity join, which finds similar objects (e.g., products, people, locations) across different sources, is a powerful tool for tackling the challenge.

For example, suppose a data scientist collects a set of restaurants from Groupon.com and would like to know which restaurants are highly rated on Yelp.com. Since a restaurant may have different representations in the two data sources (e.g., "10 East Main Street" vs. "10 E Main St., #2"), she can use similarity join to find these similar restaurant pairs and integrate the two data sources together.

Threshold-driven similarity join has been extensively studied in the past [5, 12, 7, 32, 30, 31, 25, 55, 26, 17, 24, 56, 13, 20, 9]. To use it, one has to go through three steps: (a) selecting a similarity function (e.g., Jaccard), (b) selecting a threshold (e.g., 0.8), and (c) running a similarity join algorithm to find all object pairs whose similarities are at least 0.8. The existing studies are mainly focused on Step (c). However, both Steps (a) and (b) deeply implicate humans in the loop, which can be orders of magnitude slower than conducting the actual similarity join.

One may argue that, in reality, humans are able to quickly select an appropriate similarity function and a corresponding threshold for a given similarity join task. For choosing similarity function, this may be true because humans can understand the semantics of each similarity function and choose the one that meets their needs.

However, selecting an appropriate threshold may be far from easy. It is extremely difficult for humans to figure out the effect of different thresholds on result quality. Choosing a good threshold depends on not only the specified similarity function but also the underlying data. We conduct an empirical analysis on the optimal thresholds for a diverse range of similarity join tasks, where the optimal thresholds maximize F_1-scores [27]. Table 4.1 shows the results (details of the experiment are in Section 4.4). We find that the optimal thresholds for the tasks are quite different. Even for the same similarity function, the optimal thresholds may still vary a lot. For example, the optimal threshold of a record-linkage task on the Restaurants

Table 4.1: Example of optimal thresholds w.r.t. various tasks.

Dataset	Task	Optimal Threshold	Similarity
Wiki Editors	Spell checking	0.625	Jaccard
Restaurants	Record linkage	0.6	Jaccard
Scholar-DBLP	Record linkage	0.34	Jaccard
Wiki Links	Entity matching	0.9574	Tversky

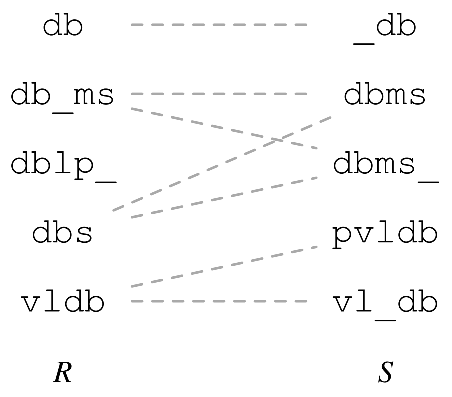
dataset is 0.6, which differs a lot from the optimal threshold of 0.34 on the Scholar-DBLP dataset using the same similarity function.

To solve this problem, one idea may be to label some data and then use the labeled data as the ground truth of matching pairs to tune the threshold. However, human labeling is error-prone and time-consuming, which significantly increases the (end-to-end) time of data integration or cleaning using similarity join.

In this chapter, we tackle this problem from a different angle – *can we achieve high-quality similarity join results without requiring humans to label any data or specifying a similarity threshold?* Our key insight is inspired by the concept of preference in areas like economics, which is an ordering of different alternatives (results) [3]. Taking Yelp.com as an example, the restaurants can be presented in different ways such as by distance, price, or rating. The different ordering may meet different search intents. A user needs to evaluate her query intent and choose the most suitable ranking accordingly. Similarly, when performing a similarity join, we seek to provide a number of *result set preferences* for a user to select from. Intuitively, a result set preference can be thought of as an objective function to capture how much a user likes a similarity join result. Once a particular preference is chosen, we automatically tune the threshold such that the preference objective function is maximized, and then we return the corresponding similarity join result. We call this new similarity join model *preference-driven similarity join.* Compared to the traditional threshold-driven similarity join, this new model does not need any labeled data.

As a proof of concept, our paper proposes two preferences from different yet complementary perspectives. The first preference MaxGroups groups the joined pairs where each group is considered as an entity across two data sources, and returns the join result having the largest number of groups. The second preference MinOutJoin balances between matching highly similar pairs and joining many possibly matching pairs, and favors the join result minimizing the outer-join size. According to our experiments on various datasets with ground-truth, the preference-driven approach can achieve optimal or nearly optimal F_1-scores on different tasks without knowing anything about the optimal thresholds.

Given a result set preference, a challenge is how to develop an efficient algorithm for preference-driven similarity join. This problem is more challenging than the traditional threshold-driven similarity join because it involves one additional step: finding the best threshold such that a preference is maximized. The brute-force method needs to compute



$\Longrightarrow$

(r, s)	$sim(r, s)$
(db_ms, dbms_)	1
(db_ms, dbms)	
(vldb, pvldb)	0.8
(vldb, vl_db)	
(dbs, dbms)	0.75
(db, _db)	
(dblp_, pvldb)	0.667
(dblp_, vl_db)	
(db_ms, _db)	
(dblp_, _db)	0.6
(dbs, dbms_)	
(db, dbms)	0.5
(dbs, _db)	

The optimal threshold is 0.75. The edges represent the ground-truth $\mathbb{C}^+$.

Figure 4.1: Example of $join(\theta)$, where $\theta = 0.5$.

the similarity values for all the pairs. It is highly inefficient even for datasets of moderate size. We solve this problem by developing a novel similarity join framework along with effective optimization techniques. The experimental results show that the proposed framework achieves several orders of magnitude speedup over the brute-force method.

The rest of the chapter is organized as follows. In Section 4.1, we formally define the problem of preference-driven similarity join. In Section 4.2, we design two result set preferences from different perspectives. In Section 4.3, we propose a preference-driven similarity join framework, and develop efficient algorithm for set-based similarity functions. In Section 4.4, we evaluate our approach on four real-world web datasets from a diverse range of applications. The results suggest that preference-driven similarity join is a promising idea to tackle the threshold selection problem for similarity join, and verify that our method is effective and efficient.

4.1 Problem Definition

Let R and S be two sets of objects, $\mathbb{C}^+$ denote the ground-truth that is the set of pairs in $R \times S$ that should be joined/matched, and $\mathbb{C}^-$ denote the remaining pairs, i.e., $\mathbb{C}^- = (R \times S) \setminus \mathbb{C}^+$. We call the pairs in $\mathbb{C}^+$ and $\mathbb{C}^-$ matching pairs and non-matching pairs, respectively. In general, $\mathbb{C}^+$ and $\mathbb{C}^-$ are assumed unknown. Figure 4.1 shows a toy running example of R and S, as well as the ground-truth $\mathbb{C}^+$.

Let $sim : (r, s) \in R \times S \to [0, 1]$ denote a similarity function.

Definition 5 (Threshold-driven similarity join)**.** *Given a similarity function sim and a threshold θ, return all the object pairs (r, s) whose similarity values are at least θ, that is,*

$$join(R, S, sim, \theta) = \{(r, s) \in R \times S \mid sim(r, s) \geq \theta\} \quad \square$$

If R, S, and sim are clear from the context, we write $join(\theta)$ for the sake of brevity.

We can regard $join(\theta)$ as a classifier, where the pairs returned by the function are the ones classified as positive, and the rest pairs $(R \times S)\backslash(join(\theta))$ classified as negative, that is, not-matching. Figure 4.1 shows an example of $join(0.5)$ using Jaccard similarity. Here, for simplicity we tokenize a string into a set of characters. For example, $jaccard(\texttt{dblp\_}, \texttt{\_db}) = \frac{|r \cap s|}{|r \cup s|} = \frac{|\{\texttt{\_},\texttt{d},\texttt{b}\}|}{|\{\texttt{\_},\texttt{d},\texttt{b},\texttt{l},\texttt{p}\}|} = \frac{3}{5} = 0.6$.

Similarity join seeks to find a threshold that leads to the best result quality. Theoretically, there are an infinite number of thresholds to choose from. However, we only need to consider a finite subset of the possible thresholds, which is the set of the similarity values of all object pairs in $R \times S$, i.e., $\{sim(r, s) \mid r \in R, s \in S\}$, because, for any threshold not in the finite set, there is always a smaller threshold in the finite set having the same join result. For example, threshold 0.9 is not in the finite set in the example in Figure 4.1 but $join(0.9) = join(0.8)$.

Threshold tuning is time consuming and labor intensive. Therefore, we develop preference-driven similarity join to overcome the limitations. Before a formal definition, we first introduce the concept of *result set preference*, to capture the user preferences on a similarity join result. Formally, a result set preference is a score function h: $(R, S, sim, \theta) \rightarrow \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers. Obviously, a result set is determined by R, S, sim and θ. The result set preference gives a score on how well the result set meets a user's preference. The higher the score, the better. If R, S, and sim are clear from the context, we write $h(\theta)$ for the sake of brevity.

Definition 6 (Preference-driven Similarity Join)**.** *Given a similarity function sim and a result set preference h, return the most preferred result $join(\theta^*)$ where θ^* is the largest threshold in the finite set maximizing h.* $\square$

For the ease of presentation, we introduce some notations. First, we denote by $join^{=}(\theta) = \{(r, s) \in join(\theta) \mid sim(r, s) = \theta\}$ the subset of joined pairs w.r.t. similarity θ. For example, in Figure 4.1, $join^{=}(0.75) = \{(\texttt{dbs}, \texttt{dbms})\}$. Second, we denote by $cover^R(\theta) = \{r \mid \exists s : (r, s) \in join(\theta)\}$ the set of objects in R that are joined when the similarity threshold is θ. Similarly, we have $cover^S(\theta) = \{s \mid \exists r : (r, s) \in join(\theta)\}$. For example, in Figure 4.1, $cover^R(0.75) = \{\texttt{db\_ms}, \texttt{dbs}, \texttt{vldb}\}$. Last, for $r \in R$, we denote by $top^S(r)$ the set of most similar object(s) in S including ties. Similarly, we have $top^R(s)$. For example, in Figure 4.1, $top^S(\texttt{vldb}) = \{\texttt{pvldb}, \texttt{vl\_db}\}$.

(db_ms, dbms_)	1	1	1
(db_ms, dbms)			
(vldb , pvldb)	0.8	2	**2**
(vldb , vl_db)			
(dbs , dbms)	0.75	2	**2**
(db , _db)			
(dblp_, pvldb)	0.667	**3**	**2**
(dblp_, vl_db)			
(db_ms, _db)			
(dblp_, _db)	0.6	1	-1
(dbs , dbms_)			
(db , dbms)	0.5	1	-3
(dbs , _db)			
…	…	…	…
$(r, s) \in R \times S$	θ	$h_c(\theta)$	$h_o(\theta)$

Figure 4.2: Example of h_c and h_o.

4.2 Two Result Set Preferences

As a proof of concept, we present two result set preferences.

4.2.1 MaxGroups: Maximum Number of Non-Trivial Connected Components

Our first preference, MaxGroups, partitions objects into different groups without any prior knowledge. For a similarity threshold θ, we construct a bipartite graph $G_\theta = \left(U^R, V^S, join(\theta)\right)$, where U^R and V^S are two disjoint sets of nodes representing the objects in R and S, respectively, and every pair in $join(\theta)$ defines an edge. As indicated in [21], each connected component in the bipartite graph is an entity, where the objects in the same connected component are the entity's different representations.

MaxGroups prefers the similarity join result with more non-trivial connected components (i.e., connected components with at least two nodes, one in R and another in S). The intuition is that heuristically we want to match as many entities as possible across R and S. Let $J(G_\theta)$ denote the set of non-trivial connected components in a bipartite G_θ, we define result set preference MaxGroups as

$$h_c(\theta) = |J(G_\theta)|$$

Figure 4.2 gives an example of h_c on our toy dataset in Figure 4.1.

Given a similarity join result $join(\theta)$, the time complexity of computing $h_c(\theta)$ is $O(|R| + |S| + |join(\theta)|)$ by simply computing the connected components of the bipartite graph G_θ.

$$\begin{array}{l} (\texttt{db\_ms}\ ,\texttt{dbms}\ ) \\ (\texttt{db\_ms}\ ,\texttt{dbms\_}) \\ (\texttt{dbs}\ ,\texttt{dbms}\ ) \\ (\texttt{vldb}\ ,\texttt{pvldb}) \\ (\texttt{vldb}\ ,\texttt{vl\_db}) \end{array} \cup \begin{array}{l} (\texttt{db}\ ,\texttt{NULL}) \\ (\texttt{dblp\_},\texttt{NULL}) \end{array} \cup (\texttt{NULL},\texttt{\_db})$$

Figure 4.3: Example of $outjoin(\theta)$, where $\theta = 0.75$.

Since we need to compute the preference scores for multiple similarity thresholds θ, there are opportunities to reduce the computational cost. We will discuss the details in Section 4.3.3.

4.2.2 MinOutJoin: Minimum Outer-Join Size

We make the following observation. On the one hand, if we set a too high similarity threshold and thus be too strict in similarity, many objects may not be matched with their counterparts due to noise. The extreme case is that, if we set the similarity threshold to 1, only those perfectly matching objects are joined. On the other hand, if we set a too low similarity threshold and thus be too loose in similarity, many not-matching objects may be joined by mistake. The extreme case is that, by setting the similarity threshold to 0, every pair of objects in the two sets are joined.

We need to find a good balance between the two and strive to a good tradeoff. Technically, full outer-join includes both joined entries and those not joined (by matching with `NULL`). The size of the full outer-join is jointly determined by the number of objects joined and the number of objects not joined. The two numbers trade off each other. Therefore, if we minimize the size of the full outer-join, we reach a tradeoff between the two ends. This is the intuition behind our second preference, MinOutJoin.

The full outer similarity join result w.r.t. a similarity threshold is

$$\begin{aligned} outjoin(\theta) = join(\theta) &\cup \left\{(r, \texttt{NULL}) \;\middle|\; r \in R \setminus cover^R(\theta)\right\} \\ &\cup \left\{(\texttt{NULL}, s) \;\middle|\; s \in S \setminus cover^S(\theta)\right\} \end{aligned}$$

where $\left\{(r, \texttt{NULL}) \;\middle|\; r \in R \setminus cover^R(\theta)\right\}$ is the set of objects in R that are not joined, and $\left\{(\texttt{NULL}, s) \;\middle|\; s \in S \setminus cover^S(\theta)\right\}$ is the set of objects in S that are not joined. Figure 4.3 illustrates an example of a full outer-join. We define our preference MinOutJoin as

$$h_o(\theta) = |R| + |S| - |outjoin(\theta)| = \left|cover^R(\theta)\right| + \left|cover^S(\theta)\right| - |join(\theta)|$$

where $|R| + |S|$ is a constant given R and S. Figure 4.2 gives an example of h_o on our toy dataset.

This preference gives a penalty when multiple objects in a set are joined with multiple objects in the other set. Joining x objects in R and y objects in S results in $x \cdot y$ pairs in the full outer-join. Not joining them results in at most $x + y$ of pairs in the full outer-join. When $x > 1$ and $y > 1$, we have $x \cdot y \geq x + y$.

Given a threshold θ, it is straightforward to compute $cover^R(\theta)$, $cover^S(\theta)$ and $join(\theta)$ according to their definitions by scanning the join result $join(\theta)$. The time complexity of computing $h_o(\theta)$ is $O(|join(\theta)|)$ for each θ. Since we need to compute the preference scores for multiple similarity thresholds θ, there are opportunities to reduce the computational cost. We will discuss the details in Section 4.3.3.

4.3 Algorithm Framework

In this section, we present an efficient framework for the preference-driven similarity join problem. A brute-force solution is to compute the similarities for all the object pairs, calculate the preference score w.r.t. each possible threshold, and return the similarity join result with the highest preference score. This brute-force method may be inefficient. Computing similarities for all pairs is often prohibitive. The number of possible thresholds can be very large, $|R| \times |S|$ in the worst case. It is crucial to reduce the cost involved in this process.

To tackle the challenges, we propose a preference-driven similarity join framework in Algorithm 2. Central to the framework are four key functions. Function PivotalThresholds (Section 4.3.1) identifies a small set of thresholds Θ, called the pivotal thresholds, that are guaranteed to cover the best preference score obtained from all the possible thresholds. Function IncrementalSimJoin (Section 4.3.2) checks the pivotal thresholds in value descending order and incrementally computes the similarity join result for each threshold. We propose a new optimization technique, called lazy evaluation, to further improve the efficiency. Function IncrementalScore (Section 4.3.3) computes the preference score for each threshold. It is possible to reduce the cost by computing the scores incrementally when checking the pivotal thresholds in value descending order. Function EarlyTermination (Section 4.3.4) determines whether we can stop checking the remaining pivotal thresholds by comparing the upper bound $\widehat{h}(\theta_i)$ with currently best score $h(\theta^*)$ once a similarity join result $join(\theta_i)$ is computed.

4.3.1 Pivotal Thresholds

Not every threshold has a chance to lead to the maximum preference score. In this section, we study how to identify a small set of thresholds Θ such that the maximum preference score can be obtained by only evaluating Θ, i.e., $\max_{\theta \in [0,1]} h(\theta) = \max_{\theta \in \Theta} h(\theta)$.

Algorithm 2: Preference-driven similarity join framework.

```
Input: objects $R$ and $S$, similarity function $sim$, preference $h$
Output: the most preferred join result $join(\theta^*)$
1 $\Theta \leftarrow$ PivotalThresholds$(R, S, sim)$
2 foreach threshold $\theta_i \in \Theta$ in descending order do
3     $join^=(\theta_i) \leftarrow$ IncrementalSimJoin$(\theta_{i-1}, \theta_i)$
4     $join(\theta_i) \leftarrow join(\theta_{i-1}) \cup join^=(\theta_i)$
5     if $h(\theta_i) > h(\theta^*)$ then $\theta^* \leftarrow \theta_i$          // IncrmentalScore
6     else if $\widehat{h}(\theta_i) \leq h(\theta^*)$ then break          // EarlyTermination
7 return $join(\theta^*)$
```

Consider $\Theta = \left\{ sim(r,s) \,\middle|\, r \in R, s \in S, r \in top^R(s) \ \wedge \ s \in top^S(r) \right\}$. Clearly, Θ is often dramatically smaller than $|R| \times |S|$, as

$$|\Theta| \leq \min \left\{ \left| \left\{ sim(r,s) \,\middle|\, r \in R, s \in S \text{ where } s \in top^S(r) \right\} \right|, \right.$$
$$\left. \left| \left\{ sim(r,s) \,\middle|\, r \in R, s \in S \text{ where } r \in top^R(s) \right\} \right| \right\} \leq \min \left\{ |R|, |S| \right\}$$

For example, in Figure 4.1, $\Theta = \{1, 0.8, 0.667\}$.

We can show that both MaxGroups and MinOutJoin have the same set of pivotal thresholds. The basic idea is that, for any $\theta \notin \Theta$, there exists $\theta' > \theta$ such that $h(\theta') \geq h(\theta)$. Remind that, through the paper, we only need to discuss the thresholds within the finite set $\{sim(r,s) \mid r \in R, s \in S\}$ as discussed in Section 4.1.

Lemma 5. *Given a threshold θ, if $r \notin top^R(s)$ or $s \notin top^S(r)$ for any $(r,s) \in join^=(\theta)$, then $\exists \theta' > \theta : h_c(\theta') \geq h_c(\theta)$.*

Proof. Let θ' be a threshold such that $join(\theta) \setminus join(\theta') = join^=(\theta)$. Bipartite G_θ can be derived by adding those new edges in $join^=(\theta)$ to bipartite $G_{\theta'}$. For any $(r,s) \in join^=(\theta)$, if $r \notin top^R(s)$, then r must be already in a non-trivial connected component of $G_{\theta'}$, and s can only be added to the non-trivial connected component where r belongs to. Similar situation happens if $s \notin top^S(r)$. Since there is not any new non-trivial connected component in G_θ comparing to $G_{\theta'}$, $h_c(\theta') \geq h_c(\theta)$. □

Lemma 6. *Given a threshold θ, if $r \notin top^R(s)$ and $s \notin top^S(r)$ for any $(r,s) \in join^=(\theta)$, then $\exists \theta' > \theta : h_o(\theta') \geq h_o(\theta)$.*

Proof. Let θ' be a threshold such that $join(\theta)\backslash join(\theta') = join^{=}(\theta)$. When $r \notin top^R(s) \vee s \notin top^S(r)$ for any $(r,s) \in join^{=}(\theta)$,

$$\begin{aligned}
&|join(\theta)| - |join(\theta')| = |join^{=}(\theta)| \\
\geq &\left|\left\{(r,s) \in join^{=}(\theta) \;\middle|\; s \in top^S(r)\right\}\right| + \left|\left\{(r,s) \in join^{=}(\theta) \;\middle|\; r \in top^R(s)\right\}\right| \\
\geq &\left|\left\{r \in cover^R(\theta) \;\middle|\; s \in S \text{ where } sim(r,s) = \theta \;\wedge\; s \in top^S(r)\right\}\right| \\
+ &\left|\left\{s \in cover^S(\theta) \;\middle|\; r \in R \text{ where } sim(r,s) = \theta \;\wedge\; r \in top^S(s)\right\}\right| \\
= &\left|cover^R(\theta)\right| - \left|cover^R(\theta')\right| + \left|cover^S(\theta)\right| - \left|cover^S(\theta')\right|
\end{aligned}$$

Thus, $h_o(\theta') - h_o(\theta) \geq 0$. □

Thanks to the existing fast top-k similarity search algorithms [31, 29], obtaining the pivotal thresholds can be efficient by computing the most similar objects of each object in R and S, respectively. According to the above lemmas, it is guaranteed that the largest threshold having the maximum score in the finite set of thresholds is always in Θ.

4.3.2 Incremental Similarity Join

In this section, we present an efficient algorithm that incrementally computes the similarity join result $join(\theta_i)$ w.r.t. threshold θ_i. We do not need to conduct a similarity join for each pivotal threshold. Since $join(\theta) = \cup_{\theta' \geq \theta} join^{=}(\theta')$, for each threshold θ, we only need to compute the respective newly joined pairs $join^{=}(\theta)$. Thus, we can enumerate the thresholds in the value descending order, and compute the respective join results incrementally.

The algorithm consists of two steps. First, the algorithm incrementally computes a set of candidate pairs $cand(\theta_i)$. Second, the algorithm evaluates the similarity of each candidate pair and returns the pairs whose similarity values are at least the threshold θ_i. While this two-step approach has been used by existing similarity join algorithms [12, 7, 32, 26], our contribution is a novel optimization technique, called lazy evaluation, which lazily evaluate the similarity of each candidate pair and reduces the cost.

We focus on set-based similarity functions in this chapter. Similar strategies can be applied to other kinds of similarity functions, like string-based or vector-based. Note that multi-set (bag) can also be used here instead of set. Table 4.2 shows the definitions of the similarity functions. Jaccard, overlap, dice, and cosine similarity are widely adopted in existing similarity join literature [12, 7, 32, 26]. In addition, we include Tversky similarity [57], which is a special asymmetric set-based similarity with different weights α and $1-\alpha$ on r and s, respectively. This similarity function is very useful in certain scenarios, like matching a text with an entity contained by the text. Given an object r as a set, we use $r[:i]$ to denote the first i elements of r assuming a global ordering of elements in the set.

Table 4.2: Summary of set-based similarity functions.

Here $bound^{min}_{i,j} = |r[:i] \cap s[:j]| \leq |r \cap s|$ and
$bound^{max}_{i,j} = |r[:i] \cap s[:j]| + \min\{|r| - i, |s| - j\} \geq |r \cap s|$.

Similarity	sim	$sim^{min}_{i,j}$	$sim^{max}_{i,j}$	t_θ
Jaccard	$\frac{\|r \cap s\|}{\|r\|+\|s\|-\|r \cap s\|}$	$\frac{bound^{min}_{i,j}}{\|r\|+\|s\|-bound^{min}_{i,j}}$	$\frac{bound^{max}_{i,j}}{\|r\|+\|s\|-bound^{max}_{i,j}}$	$\lceil \theta \cdot \|r\| \rceil$
Overlap	$\frac{\|r \cap s\|}{\max\{\|r\|,\|s\|\}}$	$\frac{bound^{min}_{i,j}}{\max\{\|r\|,\|s\|\}}$	$\frac{bound^{max}_{i,j}}{\max\{\|r\|,\|s\|\}}$	$\lceil \theta \cdot \|r\| \rceil$
Dice	$\frac{2 \cdot \|r \cap s\|}{\|r\|+\|s\|}$	$\frac{2 \cdot bound^{min}_{i,j}}{\|r\|+\|s\|}$	$\frac{2 \cdot bound^{max}_{i,j}}{\|r\|+\|s\|}$	$\left\lceil \frac{\theta}{2} \cdot \|r\| \right\rceil$
Cosine	$\frac{\|r \cap s\|}{\sqrt{\|r\| \cdot \|s\|}}$	$\frac{bound^{min}_{i,j}}{\sqrt{\|r\| \cdot \|s\|}}$	$\frac{bound^{max}_{i,j}}{\sqrt{\|r\| \cdot \|s\|}}$	$\lceil \theta^2 \cdot \|r\| \rceil$
Tversky	$\frac{\|r \cap s\|}{\alpha \cdot \|r\|+(1-\alpha) \cdot \|s\|}$	$\frac{bound^{min}_{i,j}}{\alpha \cdot \|r\|+(1-\alpha) \cdot \|s\|}$	$\frac{bound^{max}_{i,j}}{\alpha \cdot \|r\|+(1-\alpha) \cdot \|s\|}$	$\lceil \theta \cdot \alpha \cdot \|r\| \rceil$

Candidate Pair Generation

Established by prefix filtering [12], if $sim(r, s) \geq \theta$, the number of elements in the overlap of the sets $|r \cap s|$ is no fewer than an overlap threshold t_θ w.r.t. $|r|$, where the overlap thresholds for set-based similarities are shown in Table 4.2. Thus, the candidate pair generation problem is converted to how to filter out the pairs with fewer than t_θ common elements.

To filter out the pairs with less than t_θ common elements, we fix a global ordering on the elements of all the objects, and sort the elements in each object based on the ordering. Like [26], we use the inverse document frequency as the global ordering. Prefix filtering [12] establishes that if $|r \cap s| \geq t_\theta$, then $r[:\#prefix_\theta(r)] \cap s[:\#prefix_\theta(r)] \neq \emptyset$, where $\#prefix_\theta(r) = |r| - t_\theta + 1$.

Using an inverted index, we do not need to enumerate each pair (r, s) to verify whether $r[:\#prefix_\theta(r)] \cap s[:\#prefix_\theta(r)] \neq \emptyset$. An inverted index maps an element to a list of objects containing the element. After building the inverted index for S, for each $r \in R$, we only need to merge the inverted lists of the elements in $r[:\#prefix_\theta(r)]$ to retrieve each $s \in S$ such that $r[:\#prefix_\theta(r)] \cap s[:\#prefix_\theta(r)] \neq \emptyset$.

Our goal is to generate the candidate pairs for $[\theta_i, \theta_{i-1})$. We use an incremental prefix filtering approach [31] that memorizes previous results to avoid regenerating the candidate pairs for $[\theta_{i-1}, 1]$.

Lazy Evaluation

Suppose we want to check whether the similarity of a candidate pair $(r, s) \in cand(\theta_i)$ is no smaller than a threshold θ_i or not. The idea of lazy evaluation is to iteratively compute both a maximum and a minimum possible value of $sim(r, s)$, denoted by $sim^{max}(r, s)$ and $sim^{min}(r, s)$, respectively. Interestingly, both $sim^{max}(r, s)$ and $sim^{min}(r, s)$ get tighter through the process, and finally converge at $sim(r, s)$. During this process, we use the values for lazy evaluation in two ways.

Algorithm 3: Incremental similarity join.

```
Input: thresholds $\theta_{i-1}$ and $\theta_i$ where $\theta_{i-1} > \theta_i$
Output: incremental similarity join result $join^{=}(\theta_i)$
1  $join^{=}(\theta_i) \leftarrow \emptyset$
2  let $cand(\theta_i)$ be the candidate pairs for $[\theta_i, \theta_{i-1})$
3  foreach pair $(r, s) \in cand(\theta_i)$ do
4      while $sim^{min}(r, s) < \theta_i \leq sim^{max}(r, s)$ do
5          update $sim^{max}(r, s)$ and $sim^{min}(r, s)$
6      if $sim^{max}(r, s) < \theta_i$ then
7          find $\theta_j : \theta_{j-1} > sim^{max}(r, s) \geq \theta_j$ by binary search
8          add $(r, s)$ into $cand(\theta_j)$
9      else
10         add $(r, s)$ into $join^{=}(\theta_i)$
11 return $join^{=}(\theta_i)$
```

- If $sim^{max}(r, s) < \theta_i$, then $sim(r, s) < \theta_i$. Thus, it is only necessary to resume evaluating (r, s) for a future smaller threshold θ_j where $\theta_{j-1} > sim^{max}(r, s) \geq \theta_j$.

- If $sim^{min}(r, s) \geq \theta_i$, then $\theta_{i-1} > sim(r, s) \geq \theta_i$. Thus, (r, s) does not need to be fully evaluated at all.

We scan r and s iteratively together from left to right, according to the global ordering. Assuming $r[: i]$ and $s[: j]$ have been scanned, Table 4.2 shows the maximum/minimum possible values of $sim(r, s)$. Through the scanning, we iteratively update $sim^{max}(r, s)$ and $sim^{min}(r, s)$ accordingly.

The pseudo-code of the lazy evaluation-powered algorithm is shown in Algorithm 3. The algorithm first computes $cand(\theta_i)$, by generating a set of candidate pairs for $[\theta_i, \theta_{i-1})$ together with the previously postponed candidate pairs from larger thresholds. Then, the algorithm examines each candidate pair in $cand(\theta_i)$ and decides whether it should be added into $join^{=}(\theta_i)$ or postponed and added into $cand(\theta_j)$ for a smaller threshold θ_j (found by binary search over the rest of the thresholds). Finally, $join^{=}(\theta_i)$ is returned.

4.3.3 Incremental Score Computation

For both our preferences, computing the preference score for each similarity threshold is straightforward. However, as we need to compute the preference scores for multiple thresholds, it is necessary to explore how to further reduce the cost.

For preference MaxGroups, when computing the join result incrementally by decreasing θ, if two objects of each newly joined pair in $join^{=}(\theta)$ are in different connected components, the connected components are merged together to form a larger connected component. We use a disjoint-set data structure to dynamically track newly joined pairs, and update the connected components accordingly. It only takes almost $O(1)$ amortized time [15] for each newly joined pair in $join^{=}(\theta)$.

For preference MinOutJoin, when processing incrementally by the value decreasing order of θ, we only need to scan each $join^{=}(\theta)$ to update $join(\theta)$, $cover^R(\theta)$, $cover^S(\theta)$, and further the preference score. It only takes $O(1)$ time for each newly joined pair in $join^{=}(\theta)$.

4.3.4 Early Termination

The goal of early termination is to determine if we can return the current most preferred result without evaluating the remaining thresholds. In our algorithm, the thresholds are evaluated in the descending order. Suppose threshold θ_i has just been evaluated. At this point, we have known the preference score for each threshold that is at least θ_i. Let $h(\theta^*)$ denote the current best preference score, i.e., $h(\theta^*) = \max_{\theta \geq \theta_i} h(\theta)$. If we can derive an upper-bound $\widehat{h}(\theta_i)$ of the preference scores for the remaining thresholds and show that the upper-bound is no larger than $h(\theta^*)$, then it is safe to stop at θ_i and $h(\theta^*)$ is the best result overall.

For MaxGroups, as the threshold decreases, a previously unseen non-trivial connected component can only be created by merging two trivial connected components. Since a new non-trivial connected component contains at least one object from $R \setminus cover^R(\theta_i)$ and one object from $S \setminus cover^S(\theta_i)$, the number of non-trivial connected components in any $G_{\theta'}$ such that $\theta' < \theta_i$ is at most $\min\left\{\left|R \setminus cover^R(\theta_i)\right|, \left|S \setminus cover^S(\theta_i)\right|\right\}$. Thus, an upper-bound is

$$\widehat{h_c}(\theta_i) = h_c(\theta_i) + \min\left\{\left|R \setminus cover^R(\theta_i)\right|, \left|S \setminus cover^S(\theta_i)\right|\right\}$$

For MinOutJoin, as the threshold decreases, the join result $join(\theta_i)$ includes more pairs. Whenever a new pair (r, s) is joined, $|join(\theta_i)|$ increases by one. The only way to get the preference score increased by 1 is that $\left|cover^R(\theta_i)\right|$ and $\left|cover^S(\theta_i)\right|$ both increase by 1. In this case, r has to be come from $R \setminus cover^R(\theta_i)$ and s has to be come from $S \setminus cover^S(\theta_i)$. Therefore, the preference score can at most increase by $\min\left\{\left|R \setminus cover^R(\theta_i)\right|, \left|S \setminus cover^S(\theta_i)\right|\right\}$. Accordingly, we set an upper-bound to

$$\widehat{h_o}(\theta_i) = h_o(\theta_i) + \min\left\{\left|R \setminus cover^R(\theta_i)\right|, \left|S \setminus cover^S(\theta_i)\right|\right\}$$

4.4 Experimental Results

We present a series of experimental results in this section. The programs were implemented in Python running with PyPy[1]. The experiments were conducted using a Mac Pro Late 2013 Server with Intel Xeon 3.70GHz CPU, 64GB memory, and 256GB SSD.

[1]PyPy (`http://pypy.org/`) is an advanced just-in-time compiler, providing 10 times faster performance for our algorithm than the standard Python interpreter.

Table 4.3: Dataset characteristics.

Dataset	R			S			$\lvert\mathbb{C}^+\rvert$
	$\lvert R\rvert$	Max. Len.	Avg. Len.	$\lvert S\rvert$	Max. Len.	Avg. Len.	
Wiki Editors	2,239	20	9.65	1,922	16	8.66	2,455
Restaurants	533	96	48.38	331	91	43.5	112
Scholar-DBLP	64,259	259	115.9	2,562	326	106.61	5,347
Wiki Links	187,122	1,393	17.08	168,652	209	16.35	202,272

4.4.1 Datasets

We adopt four real-world web datasets with ground-truth for evaluation. Table 4.3 shows the characteristics of each dataset.

Wiki Editors [50] is about misspellings of English words, made by Wikipedia page editors where the errors are mostly typos. Each misspelling has at least one correct word. R contains the misspellings, while S contains the correct words. The ground-truth $\mathbb{C}^+$ are pairs of each misspelling and the corresponding correct word.

Restaurants [51] links the restaurant profiles between two websites. Each profile contains the name and address of a restaurant. We remove the phone number and cuisine type, which are available in the original data, to make it more challenging. R and S are profiles of restaurants, and the ground-truth $\mathbb{C}^+$ identifies the pairs of profiles linking the same restaurants. Every restaurant has at most one match.

Scholar-DBLP [52] finds the same publications in Google Scholar and DBLP, where each record in DBLP has at least one matching record in Google Scholar. Each record on both websites contains the title, author names, venue, and year. R and S are publications identified by Google Scholar and DBLP, respectively, and the ground-truth $\mathbb{C}^+$ are pairs of records linking the same publications.

Wiki Links [53] is a large dataset containing short anchor text on web pages and the Wikipedia link that each anchor text contains. R contains the anchor text, while S contains the Wikipedia entities extracted from Wikipedia links. For example, link `https://en.wikipedia.org/wiki/Harry_Potter_(character)` is converted to entity "Harry Potter". The ground-truth $\mathbb{C}^+$ are pairs linking each anchor text and its entity. Each anchor text may also contain the text of the entity that the Wikipedia link refers to, such as "…Harry Potter is a title character…". This dataset has multiple files, where each contains roughly $200,000$ mappings (about 200 MB). We used the first file in most of our evaluation except for the scalability evaluation where the first four files were used.

To adopt set-based similarities, each string needs to be converted to a set. Empirically, people often convert a long string into a bag of words and convert a short string into a set of grams. For the first dataset, since a string represents a word, we convert each string into a bag of 2-grams. For the rest three datasets, since the strings are much longer, we convert each string into a bag of words. We use Jaccard similarity for the first three datasets. For

Table 4.4: Sensitivity of thresholds on F_1-score.

Thresholds: optimal $\theta^{\#}$, higher $\theta^{+} = \min\left\{1, \theta^{\#} + 0.1\right\}$, lower $\theta^{-} = \max\left\{0, \theta^{\#} - 0.1\right\}$.

Dataset	using θ^{+}	using $\theta^{\#}$	using θ^{-}	Preference-driven
Wiki Editors	0.599	0.764	0.705	0.764 using h_c and h_o
Restaurants	0.725	0.816	0.597	0.809 using h_o
Scholar-DBLP	0.697	0.759	0.554	0.754 using h_c
Wiki Links	0.774	0.780	0.624	0.780 using h_c

Numbers in red are when ours achieves optimal F_1-scores.

the Wiki Links dataset, since anchor text often contains both the entity text and some other irrelevant texts, we choose Tversky similarity ($\alpha = 0.1$).

4.4.2 Threshold-driven vs. Preference-driven

In this section, we empirically investigate the pros and cons of both our preference-driven approach and other possible solutions for tuning a similarity join threshold.

We demonstrate the sensitivity of thresholds w.r.t. F_1-score in Table 4.4. A small deviation from the optimal threshold affects the result quality dramatically. This clearly shows that threshold tuning is crucial for threshold-driven similarity join. On two datasets, our preference-driven approach can achieve the optimal F_1-score, and on the other two datasets, they are very close to the optimal.

To compare with human labeling approaches, we adopt two supervised approaches to tune a similarity threshold. The first one is random sampling, where pairs are randomly sampled and labeled. We use the same sampling method as [54], where the larger set R is sampled with a specified sampling rate, and then joined with S to derive all the pairs to be labeled. Apparently, at least $|S|$ pairs need to be labeled. The second approach [10] uses active learning to tune a threshold by incrementally labeling the most uncertain pair to the classifier. When there is a tie, the one with the highest similarity is selected. The threshold that achieves the best on the labeled data is selected. Figure 4.4 shows the results. For random sampling, only using a very high sampling rate like 10% can almost catch up with our method. For active learning, the number of necessary labels is significantly reduced, however, still hundreds to thousands of pairs need to be labeled on most of the datasets.

The total running time of the two supervised approaches contains two parts: the labeling part which tunes the threshold, and the joining part using the tuned threshold. In comparison, our approach returns both a threshold and its join results in a unified framework. We calculate the end-to-end time that the threshold-driven approach needs in order to achieve the same quality as our preference-driven approach. For simplicity, we assume that each pair takes $1s$ to be labeled correctly by a human, which is a very conservative estimation.

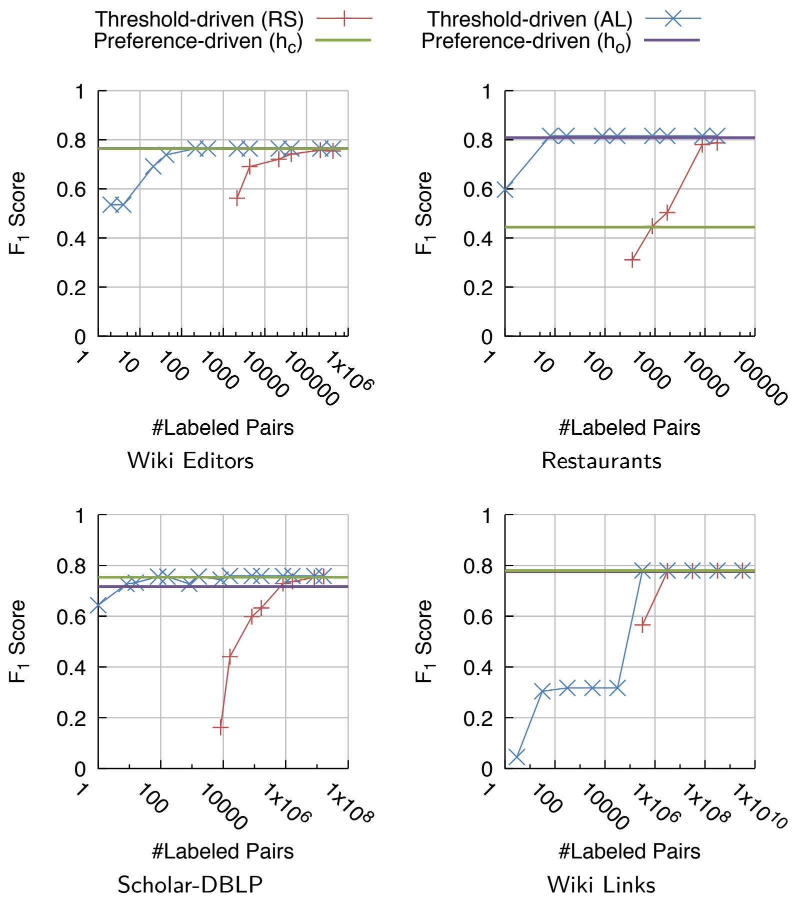
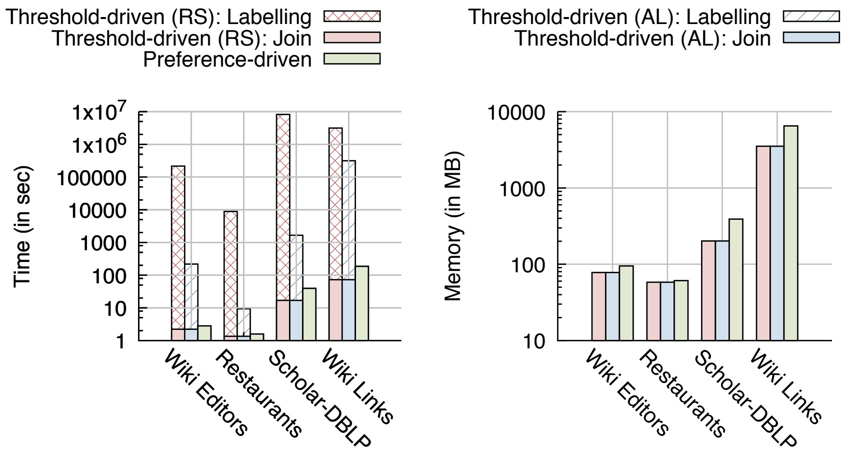


Figure 4.4: Accuracy comparisons of threshold-driven (using random sampling (RS) or active learning (AL) for threshold tuning) and preference-driven approaches.

Figure 4.5 shows the results. The labeling step is much more costly than the joining step. The end-to-end time of the preference-driven approach is orders of magnitude faster. The preference-driven approach uses almost twice of memory, due to caching candidate pairs.

4.4.3 Accuracy of Preference-driven Approach

In this section, we evaluate the accuracy of our preference-driven approach. Table 4.5 compares the results between h_c and h_o. The precision, recall, and F_1-score are presented, together with the preferred threshold θ^*. For Wiki Editors, due to its nature of high similarity between misspellings and correct words, both preferences return the same optimal result. For the record-linkage task on Restaurants, h_o gives the significantly better result as it favors one-to-one matching. For Scholar-DBLP, h_c gives the better result, because maximizing the number of non-trivial connected components actually satisfies the nature of many-to-many matching. For the Wiki Links, h_c gives the optimal result, and h_o is quite close.



(a) Running time. (b) Peak memory.

For threshold-driven approach, we tune the threshold such that it achieves the closest (within 0.01) F_1-score as preference-driven approach. We assume that each pair needs $1s$ to label. The same preference in Table 4.4 is used here for each dataset.

Figure 4.5: Efficiency comparisons of threshold-driven and preference-driven approaches.

Table 4.5: Accuracy for varying datasets and preferences.

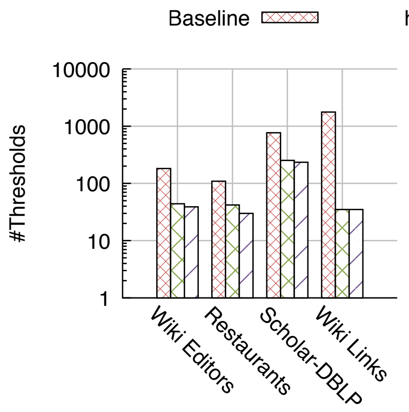
Dataset	h_c (MaxGroups)				h_o (MinOutJoin)			
Wiki Editors	0.625	0.837	0.704	**0.764**	0.625	0.837	0.704	**0.764**
Restaurants	0.429	0.291	0.938	0.444	0.556	0.805	0.813	**0.809**
Scholar-DBLP	0.361	0.841	0.683	**0.754**	0.419	0.903	0.595	0.717
Wiki Links	0.957	0.959	0.657	**0.780**	0.972	0.968	0.648	0.776
	θ^*	Precision	Recall	F_1	θ^*	Precision	Recall	F_1

4.4.4 Efficiency of Preference-driven Approach

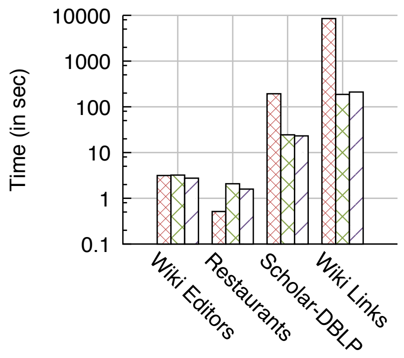
There is no existing work solving the same problem. We choose the brute-force method in Section 4.3 as a baseline to evaluate the efficiency. This method takes almost the same time and memory regardless of the preference due to the same amortized time for processing each newly joined pair incrementally.

Figure 4.6(a) shows the number of thresholds evaluated by the baseline and our algorithm. On most of the datasets, our algorithm evaluates 10 to 100 times less thresholds than the baseline. Our method achieves a significant speedup due to the combination of other optimization techniques (i.e., incremental similarity join and early termination). As Figure 4.6(b) shows, our algorithm is 10 to 100 times faster than the baseline on all the large datasets. Figure 4.6(c) shows that our method consumes significantly less memory.

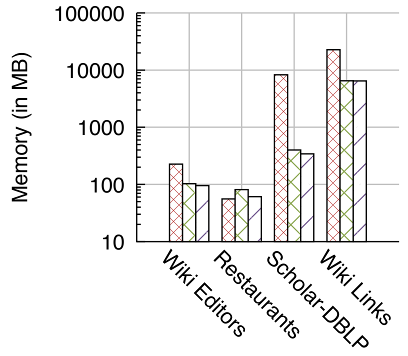
For incremental similarity join, we use lazy evaluation for speedup. Instead, a simple approach computes the exact similarities for all the new candidate pairs produced by each threshold, and puts them into a max-heap. Those pairs in the heap whose similarities are no less than the current threshold are popped out for evaluation. On smaller datasets Wiki



(a) Number of evaluated thresholds.



(b) Running time.



(c) Peak memory.

Figure 4.6: Efficiency of preference-driven approach.

Editors and Restaurants, the simple approach works slightly better, as there are not many pairs that need lazy evaluation. However, on larger datasets Scholar-DBLP and Wiki Links, our lazy evaluation approach is 2 to 5 times faster.

Scalability

For scalability, we evaluate our algorithm using h_c on a larger version of Wiki Links that contains 4 files. There are overlapping objects between different parts. Figure 4.7 shows our method's good scalability results on number of evaluated thresholds, running time, and peak memory usage.

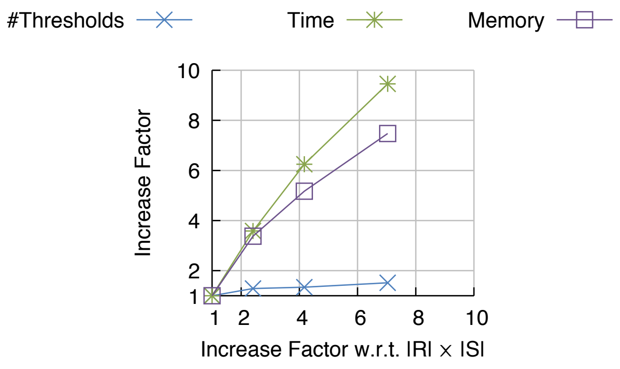


Figure 4.7: Scalability on Wiki Links using h_c.

Chapter 5

Conclusions

5.1 Conclusions

In this thesis, we first 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.

Further, we tackled the problem of similarity join based on user preference. Usually neglected in the similarity join literature, threshold selection can actually be a bottleneck in an end-to-end similarity join process. To mitigate the challenge, we propose and formalize preference-driven similarity join. We present two specific preferences as proofs of concept and develop a general framework for efficient computation methods. We evaluate our approach on four real-world datasets from a diverse range of application scenarios. The results demonstrate that preference-driven similarity join can achieves high-quality results without any labeled data, and our proposed framework is efficient and scalable.

5.2 Future Directions

As we studied, past works have extensively explored the data mapping tasks from different angles. However, we believe there are still many pieces to be filled for the puzzle of data mapping. In this section, we discuss some future directions which we believe are interesting and promising.

Traditionally, schema matching is conducted over the relational model and the tree-based model. In recent years, NoSQL databases have gained quite a popularity among users, where there is no schema or constraint enforced regarding data. Thus, even data from the same source may not share the same schema. How to effectively integrate this type of data remains an open problem, as we may have to transform and unify the schemas within the same source first. We have not seen any work regarding this direction yet. However, with the rapid growth of the NoSQL usage, this problem becomes inevitable.

Another interesting direction to be considered is transfer learning, which solves one problem and applies the gained knowledge to a related problem. For example, can we apply a model learned when integrating the student records to integrating the employee records? Admittedly, this kind of techniques are very tricky to be devised and deployed. However, we do believe this direction has great potentials, as integrating data within related domains do share great similarities.

Scalability is always a direction for future improvements. For example, given millions of web tables, can we still conduct the data integration tasks, which are usually conducted over only small data like several tables? While there are works like [64] applying schema matching over millions of web tables right now, they only compute and index pair-wise schema matching results. Traditional methods are likely not going to work well. How to scale up is still a challenging problem. Even further, more and more data are generated and collected every day. How to incrementally incorporate them into existing results is also tricky to be solved.

Bibliography

[1] A. C. Acar and Amihai Motro. Efficient discovery of join plans in schemaless data. In *IDEAS*, 2009.

[2] B. Ahmadi, M. Hadjieleftheriou, T. Seidl, et al. Type-based categorization of relational attributes. In *EDBT*, 2009.

[3] K. J. Arrow, M. D. Intriligator, W. Hildenbrand, and H. Sonnenschein. Handbook of mathematical economics. *Handbook of mathematical economics*, 1991.

[4] B. Alexe, M. Roth, and W. Tan. Preference-aware Integration of Temporal Data. *PVLDB*, 8(4):365–376, 2014.

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

[6] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD*, 2010.

[7] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.

[8] A. Bilke and F. Naumann. Schema matching using duplicates. In *ICDE*, 2005.

[9] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, 2012.

[10] L. Büch and A. Andrzejak. Approximate string matching by end-users using active learning. In *CIKM*, 2015.

[11] S. Chaudhuri, B. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *VLDB*, 2007.

[12] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[13] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, 2014.

[14] H. Elmeleegy, J. Madhavan, and A. Y. Halevy Harvesting Relational Tables from Lists on the Web. *PVLDB*, 2(1), 2009.

[15] D. Eppstein, Z. Galil, and G. F. Italiano, Dynamic graph algorithms. 1998.

[16] J. Gemmell, B. I. Rubinstein, and A. K. Chandra. Improving entity resolution with global constraints. *arXiv*, 1108.6016, 2011.

[17] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[18] J. Kang and J. F. Naughton. On schema matching with opaque column names and data values. In *SIGMOD*, 2003.

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

[20] J. Lu, C. Lin, W. Wang, C. Li, and Haiyong Wang. String similarity measures and joins with synonyms. In *SIGMOD*, 2013.

[21] A. E. Monge and C. Elkan. An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records. In *DMKD*, 1997.

[22] K. Stefanidis, M. Drosou, and E. Pitoura. PerK: personalized keyword search in relational databases through preferences. In *EDBT*, 2010.

[23] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *TODS*, 36(3):19, 2011.

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

[25] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

[26] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, 2012.

[27] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 4(10):622–633, 2011.

[28] R. H. Warren and F. W. Tompa. Multi-column Substring Matching for Database Schema Translation. In *VLDB*, 2006.

[29] X. Xu, C. Gao, J. Pei, K. Wang, and A. Al-Barakati. Continuous similarity search for evolving queries. *Knowl. Inf. Syst.*, 48(3):649–678, 2016.

[30] 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.

[31] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, 2009.

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

[33] X. Yuan, X. Cai, et al. Efficient Foreign Key Discovery Based on Nearest Neighbor Search. In *WAIM*, 2015.

[34] C. J. Zhang, L. Chen, H. V. Jagadish, et al. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 6(9), 2013.

[35] C. J. Zhang, Z. Zhao, L. Chen, et al. Crowdmatcher: crowd-assisted schema matching. In *SIGMOD*, 2014.

[36] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, et al. On multi-column foreign key discovery. *PVLDB*, 3(1), 2010.

[37] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, et al. Automatic discovery of attributes in relational databases. In *SIGMOD*, 2011.

[38] TPC-H Benchmark. `http://www.tpc.org/tpch`.

[39] MySQL Employee Database. `http://dev.mysql.com/doc/employee/en`.

[40] IMDb Alternative Interfaces. `http://www.imdb.com/interfaces`.

[41] Yelp Dataset Challenge. `http://www.yelp.ca/dataset_challenge`.

[42] L. Caruccio, G. Polese, and G. Tortora. Synchronization of Queries and Views Upon Schema Evolutions: A Survey. *ACM TODS*, 41(2), 2016.

[43] Z. Chen, V. R. Narasayya, and S. Chaudhuri. Fast Foreign-Key Detection in Microsoft SQL Server PowerPivot for Excel. *PVLDB*, 7(13), 2014.

[44] O. Hassanzadeh, K. Q. Pu, S. H. Yeganeh, R. J. Miller, L. Popa, M. A. Hernández, and H. Ho. Discovering Linkage Points over Web Data. *PVLDB*, 6(6), 2013.

[45] C. Gao, J. Pei, J. Wang, and Y. Chang. Schemaless Join for Result Set Preferences. In *IRI*, 2017.

[46] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4), 2001.

[47] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *WebDB*, 2009.

[48] R. Rymon. Search through systematic set enumeration. In *KR*, 1992.

[49] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. GORDIAN: efficient and scalable discovery of composite keys. In *VLDB*, 2006.

[50] `http://www.dcs.bbk.ac.uk/~ROGER/corpora.html`.

[51] `http://www.cs.utexas.edu/users/ml/riddle/data.html`.

[52] `http://dbs.uni-leipzig.de/en/research/projects/object_matching`.

[53] `http://www.iesl.cs.umass.edu/data/wiki-links`.

[54] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. W. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.

[55] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[56] 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.

[57] A. Tversky. Features of similarity. *Psychological Review*, 84(4):327–352, 1977.

[58] C. Gao, J. Wang, J. Pei, R. Li, and Y. Chang. Preference-driven Similarity Join. In *WI*, 2017.

[59] Nicola Perra and Santo Fortunato. Spectral centrality measures in complex networks. *Physical Review E*, 78(3), 2008.

[60] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

[61] Yeye He, Kris Ganjam, and Xu Chu. SEMA-JOIN: joining semantically-related tables using big table corpora. *PVLDB*, 8(12):1358–1369, 2015.

[62] Michael J. Cafarella, Alon Y. Halevy, and Nodira Khoussainova. Data integration for the relational web. *PVLDB*, 2(1):1090–1101, 2009.

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

[64] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*, 2012.

[65] Meihui Zhang and Kaushik Chakrabarti. Infogather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In *SIGMOD*, 2013.