SCALABLE MINING AND LINK ANALYSIS ACROSS MULTIPLE DATABASE RELATIONS

BY

XIAOXIN YIN

B.E., Tsinghua University, 2001
M.S., University of Illinois at Urbana-Champaign, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

Relational databases are the most popular repository for structured data, and are thus one of the richest sources of knowledge in the world. In a relational database, multiple relations are linked together via entity-relationship links. Unfortunately, most existing data mining approaches can only handle data stored in single tables, and cannot be applied to relational databases. Therefore, it is an urgent task to design data mining approaches that can discover knowledge from multi-relational data.

In this thesis we study three most important data mining tasks in multi-relational environments: classification, clustering, and duplicate detection. Since information is widely spread across multiple relations, the most crucial and common challenge in multi-relational data mining is how to utilize the relational information linked with each object. We rely on two types of information, — neighbor tuples and linkages between objects, to analyze the properties of objects and relationships among them.

Because of the complexity of multi-relational data, efficiency and scalability are two major concerns in multi-relational data mining. In this thesis we propose scalable and accurate approaches for each data mining task studied. In order to achieve high efficiency and scalability, the approaches utilize novel techniques for virtually joining different relations, single-scan algorithms, and multi-resolutional data structures to dramatically reduce computational costs. Our experiments show that our approaches are highly efficient and scalable, and also achieve high accuracies in multi-relational data mining.

To my dear wife Wen,

for her love, encouragement, and support.

Acknowledgments

It is amazing to look back and see how things have changed in the past several years. A new graduate student with very little research experiences, as I was five years ago, has become a Ph.D. candidate capable of performing research independently on newly emerging data mining issues. I wish to express my deepest gratitude to my advisor Dr. Jiawei Han, who made these changes possible. As an advisor, he has given me endless encouragement and support by sharing his knowledge and experiences. More importantly, he taught me to choose worthwhile topics and think deep, from which I will benefit in the rest of my career.

I am very thankful to Dr. Philip S. Yu, for the numerous insightful advices he gave me in our many discussions. He has provided great help to improve the quality of my research and this thesis.

I am also grateful to Dr. Jiong Yang for his contribution to various projects I have participated. I greatly enjoyed his ideas and his enthusiasm.

Special thanks to Dr. Kevin Chang, Dr. Marianne Winslett, Dr. Chengxiang Zhai, and Dr. Anhai Doan for their invaluable comments and suggestions to my research, which helped to keep my research work on the right track.

I would also like to express my appreciation to my friends who have contributed suggestions and feedback to my work, even if they were not officially affiliated with it: Xifeng Yan, Hwanjo Yu, Xiaolei Li, Yifan Li, Dong Xin, Zheng Shao, Dr. Hongyan Liu, Dr. Jianlin Feng, Hong Cheng, Hector Gonzalez, Deng Cai, Chao Liu. I learned a great deal from discussing with them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data mining is the process of discovering knowledge from data. In the last decade there have been great advances in data mining research, and many data mining approaches have been invented for real-life applications such as market basket analysis, direct marketing, and fraud detection. Data mining is also providing many useful methods for other disciplines such as insurance, civil engineering, and bioinformatics.

Most existing data mining algorithms (including algorithms for classification, clustering, association analysis, outlier detection, etc.) work on single tables. For example, a typical classification algorithm (e.g., C4.5 [95] or SVM [19]) works on a table containing many tuples, each of which has a class label, and a value on each attribute in the table.

It is doubtless that well-formatted, highly regular tables are easy to model and to analyze. Unfortunately, most information in the world can hardly be represented by such independent tables. In a real-world dataset there are usually many types of objects, which are linked together through different types of linkages. Such data is usually stored in relational databases or, sometimes, in XML files (which can often be converted into relational formats). For example, a computer science department may store its information in a database containing relations of professors, students, courses, course registrations, research groups, publications, etc. A simple schema of such a database is shown in Figure 1.1, and a real database will have a much more complicated schema.

This thesis studies data mining in relational databases. A relational database contains multiple interconnected relations, each of which represents a certain kind of objects or a type of relationships. As mentioned above, most existing data mining algorithms cannot be applied to relational data, unless the relational data is first converted into a single table. Unfortunately such conversion is seldom an easy task. In order to keep the useful and necessary information in data conversion, one needs to be familiar with both relational database systems and a particular database. Such data conversions are usually task specific, since different data mining tasks usually require different information. Moreover, even with good expertise, much information may still be lost during the process of compressing relational data into a table, especially the linkages between objects which cannot be represented by flat tables.

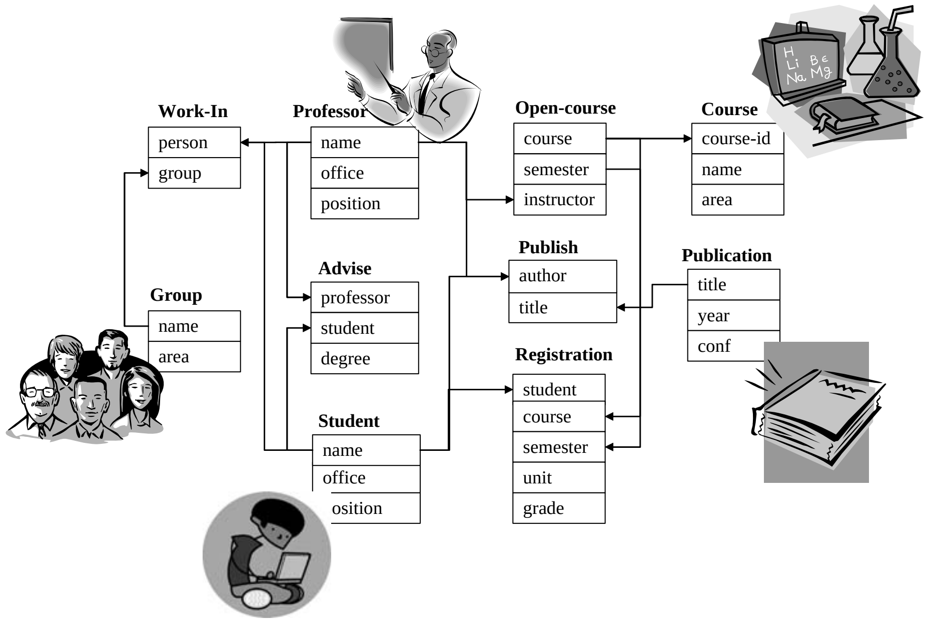


Figure 1.1: The schema of the database of a computer science department

On the other hand, a multi-relational database can often provide much richer information for data mining, and thus multi-relational data mining approaches can often achieve better performances than single-table methods. For example, in the database shown in Figure 1.1, each student is associated with various types of information in different relations, such as their courses, advisors, research groups, publications. Moreover, the objects linked with students are also linked with each other. This rich information source provides us countless opportunities of data mining. For example, we can classify students according to their academic performances, cluster students based on their research, find patterns/correlations of course registrations and publications, or detect duplicate entries among authors of publications. Because of the high popularity of relational databases, multi-relational data mining can be used in many disciplines, such as financial decision making, direct marketing, and customer relationship management.

Although very useful, multi-relational data mining faces two major challenges. First, it is much more difficult to model multi-relational data. Unlike tuples in a single table which can be modelled by vectors, multi-relational data contains heterogeneous objects and relationships among them, and there has not been widely accepted model for mining such data.

Second, many data mining approaches (e.g., classification) aim at finding a model (or hypothesis) that fits the data. In a relational database, the number of possible models is much larger than that in a single table. For example, in rule-based multi-relational classification, each rule is associated with a join path. If each relation is joinable with two other relations on average, there are $2^{k+1} - 1$ join paths of length no

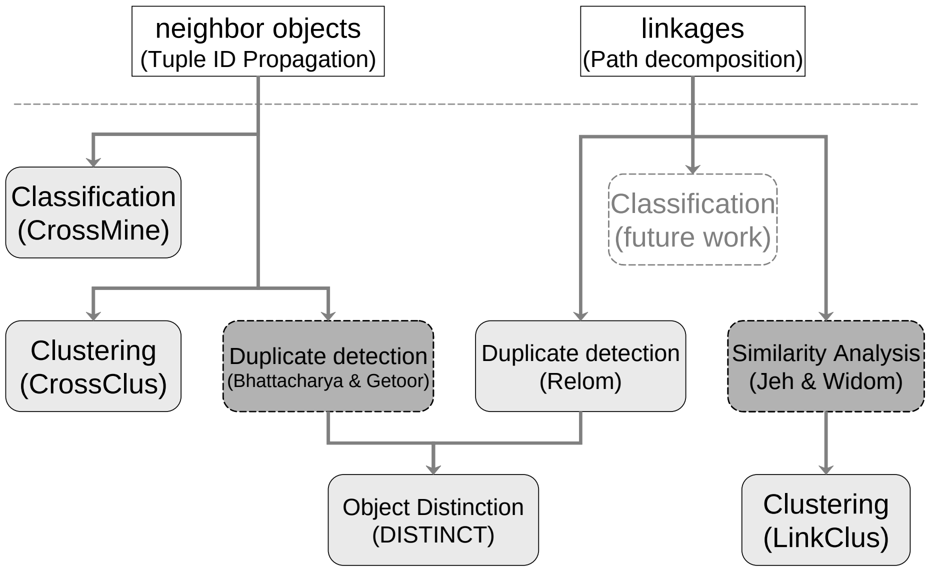


Figure 1.2: A road map of our work in multi-relational data mining

greater than k. Thus it is more complicated or at least more time consuming to search for good models in relational databases than in single tables.

The main purpose of this thesis is to study the application of data mining technology in multi-relational environments. Because most real-world relational databases have complicated schemas and contain huge amount of data, efficiency and scalability become our major concerns as well as the accuracy and effectiveness of the algorithms. In this thesis we make step-by-step developments for multi-relational data mining. Figure 1.2 shows a road map of our work. There are mainly two types of multi-relational information that are widely used for data mining: Neighbor objects (objects linked to each object through certain join paths) and linkages between objects. These two types of information are complementary for each other, as neighbor objects represent the contexts of objects, and linkages indicate relationships between objects. We propose new methods for efficiently acquiring both types of information, which are *Tuple ID Propagation* in Chapter 3 for finding neighbor objects in each relation, and *Path decomposition* in Chapter 5 for finding all linkages between two objects efficiently.

Based on Tuple ID Propagation, we propose CrossMine (Chapter 3), a highly efficient and scalable approach for multi-relational classification. Then we move to multi-relational clustering, which is a more sophisticated problem as no pre-defined classes are given. We propose CrossClus (Chapter 4), which can utilize user guidance and perform efficient clustering. Based on linkage information, we propose Relom (Chapter 5), a new approach for duplicate detection using linkages between objects. Enlightened by the pioneer work by Jeh and Widom, we invent LinkClus (Chapter 7), a new approach for linkage-based similarity analysis and clustering. Because neighbor objects and linkages are two complementary types of information, it is very helpful to combine both of them in data mining tasks. In Chapter 6 we propose DISTINCT, the first approach for distinguishing objects with identical names, which utilizes both types of information.

Specifically, the following contributions are made in this thesis.

- **Tuple ID Propagation** (Chapter 3)

  Many existing multi-relational data mining approaches suffer from efficiency and scalability problems because they repeatedly join different relations. There is often a large portion of repeated computation in these join operations. In order to address this problem, we propose *tuple ID propagation*, a method for virtually joining different relations and avoiding repeated computation. Many multi-relational mining tasks involve a *target relation*, which contains *target tuples* that are the targets for the mining task (e.g., classification, clustering, duplicate detection). Tuple ID propagation propagates the IDs of target tuples to non-target relations, so that one can easily find tuples joinable with each target tuple. This method is also very flexible, as IDs can be easily propagated between any two relations, and each relation can be associated with multiple sets of IDs. In general, tuple ID propagation gives us significant help in making our approaches efficient and scalable.

- **Efficient Multi-relational Classification** (Chapter 3)

  We start from multi-relational classification, which aims at classifying data objects in one relation using information in other relations. This is one of the simplest data mining tasks for relational data, because it has a well-defined goal with clear semantics. With the help of tuple ID propagation and adoption of divide-and-conquer strategy, we propose CrossMine, a scalable and accurate approach for multi-relational classification. It uses two algorithms for classification, one being rule-based and the other decision-tree based. The experimental results show that CrossMine is tens or hundreds of times faster than existing approaches, and achieves higher accuracy.

- **Multi-relational Clustering with User's Guidance** (Chapter 4)

  After studying classification, we turn our focus to clustering, where there is no pre-defined class labels and one has to discover relationships among objects. Unlike clustering in a single table where every attribute is considered to be pertinent, a relational database usually contains information of many aspects, and in most cases only a small part of information is relevant to the clustering task. For example, in the database in Figure 1.1 a user may want to cluster students according to their research areas, instead of their demographic information.

In multi-relational clustering it is crucial to let the user indicate her clustering goal, while it is very unlikely that the user can specify all pertinent attributes. Therefore, we allow the user to provide a simple guidance, which is one or a few pertinent attributes in the relational database. We design a methodology called CrossClus, which selects pertinent attributes across different relations by observing whether these attributes group objects in similar ways as the user guidance.

- **Multi-relational Duplication Detection** (Chapter 5)

  With the similarity analysis innovated by multi-relational clustering, we go one step further to detect duplicate objects in relational databases. After analyzing the linkages among objects in relational databases, we found that different references to the same object (e.g., a person or a conference) tend to be much more intensively connected than references to different objects. Based on this observation, we design similarity measures for evaluating how likely two objects are likely to be duplicates. The experiments show that our approach achieves high accuracy in duplicate detection.

  One crucial issue in the above method is how to efficiently find all linkages between two objects up to a maximum length. We design *path decomposition*, a method that only requires propagating information for half of that maximum length, which greatly improves the efficiency of our approach.

- **Object Distinction in Relational Databases** (Chapter 6)

  In many real databases there are different objects with identical names, such as authors with identical names in a publication database. Distinguishing objects with identical names is a different problem compared with duplicate detection, as there is very limited information associated with each reference to the name, and one needs to group the many references with identical names into clusters, so that each cluster corresponds to a real object. We use a hybrid similarity measure which combines both the context of references and linkages between them to measure their similarities. Experiments show that our approach can successfully distinguish different objects with identical names.

- **Link-based Similarity Analysis** (Chapter 7)

  In some studies discussed above, we analyze the similarity between objects using their *neighbor objects*, i.e., objects in other relations that are linked with the objects being analyzed. However, the neighbor objects are also related to each other and should not be treated separately. Thus we go one step further to perform link-based analysis on the similarity between objects using the similarities between their neighbor objects. We design hierarchical data structures that store the significant similarities and compress insignificant ones, which greatly improve the efficiency and scalability of link-based similarity analysis.

We conclude this thesis by summarizing our contributions and analyzing future directions in Chapter 8.

Chapter 2

A Survey on Multi-relational Data Mining

As discussed in Chapter 1, most existing data mining algorithms operate on a single table, while most real world databases store information in relational format. In general, multi-relational data mining is a new field and attracts significant attention from computer scientists in the past several years. In this chapter I will make an overview of research on multi-relational data mining.

2.1 Approaches for Multi-relational Data Mining

Multi-relational data mining has a precursor in the field of inductive logic programming (ILP) [38]. In recent years it has been approached from different angles, including adapting traditional feature-based data mining approaches, applying probabilistic analysis, and creating new linkage-based approaches. In this chapter I will introduce each category of approaches, and then discuss the applications of multi-relational data mining.

2.1.1 Inductive Logic Programming

Inductive Logic Programming aims at inductively learning relational descriptions in the form of logic programs. Take classification as an example, given background knowledge B, a set of positive examples P, and a set of negative examples N, an ILP approach aims at inductively finding a hypothesis H, which is a set of Horn clauses such that: (1) $\forall p \in P : H \cup B \models p$ (completeness), and (2) $\forall n \in N : H \cup B \not\models n$ (consistency).

The well known ILP classification approaches include FOIL [96], Golem [82], and Progol [81]. FOIL is a top-down learner, which builds clauses that cover many positive examples and few negative ones. Golem is a bottom-up learner, which performs generalizations from the most specific clauses. Progol uses a combined search strategy. The above three approaches are all rule-based and learn hypotheses that contain set of rules. Some more recent ILP classification approaches inductively construct decision trees, such as TILDE [13], Mr-SMOTI [5], and RPTs [83].

Besides classification, ILP has also been used in similarity analysis and clustering. In [39] an logic-based approach is proposed to measure the similarity between two objects based on their neighbor objects in a relational environment. Based on this similarity measure, ILP approaches are proposed for clustering objects in relational environments, including hierarchical clustering [67] and k-means clustering [68].

One most important factor that limits the applicability of ILP approaches is scalability issue [35, 111]. Because of the complexity in the procedure of hypothesis search, most ILP algorithms are not highly scalable with respect to the number of relations and attributes in the database, and thus are very expensive for databases with complex schemas. Many ILP approaches create physical joins of different relations when evaluating different hypothesis that involve multiple relations, which is very expensive because the joined relation is often much larger than the original relations.

2.1.2 Association Rule Mining

Association rule mining [3, 57] is by far the most studied topic in data mining, and has been applied in numerous applications such as market analysis and computational biology. Frequent pattern mining and association rule mining have also been studied in the multi-relational environments [28, 29, 90], in which a frequent pattern is defined a frequent substructure, with each node in the substructure being either a constant or a variable. However, because of complexity of relational data and the high computational cost of θ-subsumption test (*i.e.*, whether two substructures are consistent with each other), multi-relational association rule mining still remains as an expensive job.

2.1.3 Feature-based Approaches

Given the variety of data mining approaches that work on objects with multiple attributes stored in a single table, it is comparatively easy to adapt such approaches to multi-relational environments by designing feature-based multi-relational data mining approaches. Such approaches use a certain type of joined objects (or their attributes) as a feature, and modify traditional data mining approaches to adapt to such features.

In [8, 9] the authors propose approaches for duplicate detection in relational environments, which consider the neighbor objects of each object as features, and use such features to detect duplicates. In [112] an approach is proposed for clustering objects in relational databases, which treats each type of linked objects as a feature and uses such features for clustering.

2.1.4 Linkage-based Approaches

In the two pieces of milestone work on web search, — PageRank [92] and Authority-Hub Analysis [70], linkages between web pages play the most crucial roles. Linkage information is also very important in multi-relational data mining, because relational databases contain rich linkage information.

In [62] the authors propose SimRank, an approach that infers similarities between objects purely based on the inter-object linkages in a relational database. This idea is further developed in [113], which presents a scalable approach for linkage-based similarity analysis.

In [64] an approach is proposed for detecting duplicate objects by analyzing the linkages between objects. It uses random walk to model the strength of connections between objects, and conclude that two objects are likely to be duplicates if they share similar names and are strongly connected. This work illustrates the dependency between linkages and object identities.

2.1.5 Probabilistic Approaches

Bayesian Networks [77] has long been used for modeling relationships and influences among a large group of objects, and it has also been applied in relational environments. Probabilistic Relational Models (PRMs) [44, 102] is an extension of Bayesian networks for handling relational data. It can integrate the advantages of both logical and probabilistic approaches for knowledge representation and reasoning. PRMs has been used in different data mining and database applications, including classification and clustering [102] and selectivity estimation [51].

In [99] the authors use Markov Random Fields to model the relationships among objects in relational databases, in order to detect duplicate objects. The Markov Random Fields can model the interaction between the relationships between different pairs of objects. For example, two books may be duplicates if their authors are the same person, and two persons may be duplicates if they wrote the same book. This work is a typical example of applying sophisticated statistical models in relational data mining, and it may lead to more work in the same direction.

2.2 Applications of Multi-relational Data Mining

Because of the high popularity of relational database systems, multi-relational data mining can be applied in many fields with numerous applications. Here I list several major fields.

2.2.1 Financial Applications

Because most financial organizations (banks, credit card companies, *etc.*) store their information in relational formats, multi-relational data mining is often required to provide business intelligence for them. For example, multi-relational classification [96, 111] can be used for automatically handling loan applications or credit card applications, which has been studied in the discovery challenge of the PKDD conference in 1999 [93].

Another major application of data mining in finance is fraud detection. Some traditional data mining and machine learning approaches have been widely used in detecting credit card fraud, such as neural networks [18] and rule-based classification [41]. In recent years multi-relational data mining has also been used in fraud detection [84].

2.2.2 Computational Biology

Since biological datasets are often multi-relational, multi-relational data mining has many potential applications in computational biology. In [91] the authors provide a survey on this topic, and here I give a few examples. ILP has been used in prediction in molecular biology [100]. Multi-relational frequent pattern mining is used in [30] to find frequent structures in chemical compounds. Relational Bayesian networks is used in [16] for gene prediction. In general this is a fast growing research area and we can expect to see much more work in future.

2.2.3 Recommendation Systems

Recommendation systems are widely used by online retailers (*e.g.*, Amazon.com and Buy.com) for recommending products to potential buyers. Collaborative filtering [17] has been the most popular approach to this problem. Because both the product catalog data and purchase history data are stored in relational databases, multi-relational data mining has many applications in their recommendation systems. In [50] and [87] Probabilistic Relational Models is used to improve collaborative filtering. It is also promising to apply similarity analysis [62] to better model the similarity between products and customers, in order to make better recommendations.

2.3 Summary

As discussed above, there have been many studies on different approaches of multi-relational data mining, and they have been used in many fields including bioinformatics and *e*-business. On the other hand, there is also growing interests in the software industry, such as the multi-relational data mining functions in Microsoft SQL Server 2005 [27]. The vast amount of relational data in the real world provides countless opportunities for various types of multi-relational data mining, and I believe it will undergo greater development in the coming years.

Chapter 3

Efficient Multi-relational Classification

Compared with many other multi-relational data mining problems, multi-relational classification is a well-defined task. It has a clear goal, — building a model for predicting class labels of target objects using information in multiple relations, and the goal is represented explicitly by the class labels in the training set. Therefore, multi-relational classification is chosen as the first topic in this thesis.

3.1 Overview

Multi-relational classification aims at building a classification model that utilizes information in different relations. A database for multi-relational classification consists of a set of relations, one of which is the *target relation* R_t, whose tuples are called *target tuples* and are associated with class labels. (We will use "target tuple" and "target object" exchangeably, as each target tuple represents a target object.) The other relations are *non-target relations.* Figure 3.1 shows an example database of a bank, which contains information about loan applications, clients, credit cards, transactions, etc. The underlined attributes are primary-keys, and arrows go from primary-keys to corresponding foreign-keys [47]. The target relation is *Loan*. Each target tuple is either positive or negative, indicating whether the loan is paid on time. The following types of joins are considered as bridges between different relations: (1) join between a primary key k and some foreign key pointing to k, and (2) join between two foreign keys k_1 and k_2, which point to the same primary key k. (For example, the join between Loan.account-id and Order.account-id.) We ignore other possible joins because they do not represent strong semantic relationships between entities in the database.

Rule-based classifier is the most popular approach in multi-relational classification. Such a classifier is a set of rules, each having a list of predicates that involve attributes in different relations. Rule-based classification is suitable for multi-relational environments, because one can search across different relations for useful predicates, and build rules at the same time.

The study of rule-based classification has been focused on *Inductive Logic Programming (ILP).* There have been many ILP approaches for multi-relational classification [13, 14, 15, 81, 82, 83, 96]. In order to identify

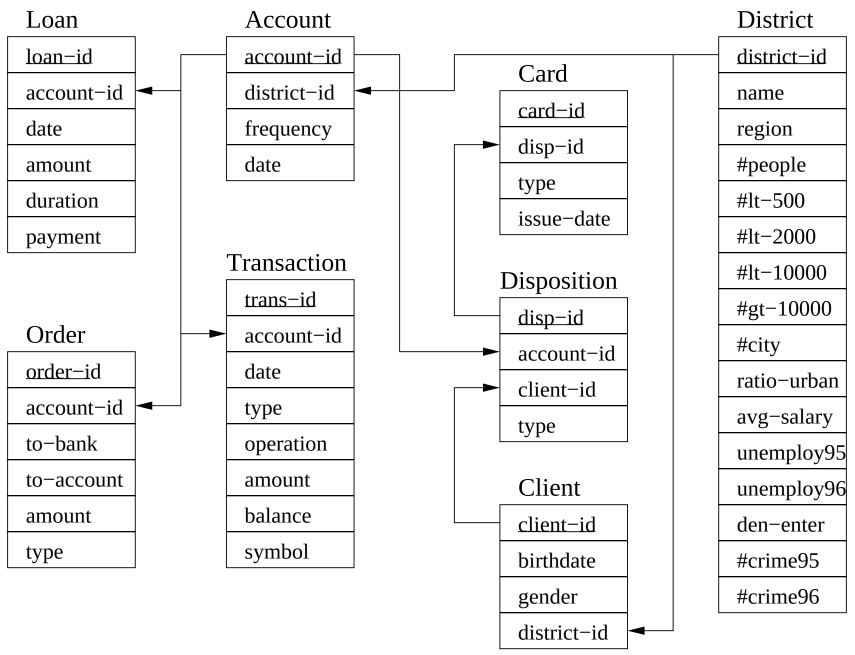


Figure 3.1: The schema of a financial database (from PKDD CUP 1999)

good predicates, most ILP approaches repeatedly join the relations along different join paths and evaluate predicates in different relations. This is very time consuming, especially when the joined relation contains many more tuples than the target relation. Therefore, although ILP approaches achieve high accuracy, most of them are not scalable w.r.t. the number of relations and the number of attributes in databases, thus are usually inefficient for databases with complex schemas.

From the above analysis we can see that the biggest challenge in designing scalable multi-relational classifiers is how to combine the information in different relations in an efficient way. We find that it is seldom necessary to physically join different relations when searching for good predicates. Instead, we can pass the most essential information for classification, — the IDs and class labels of target tuples, to the other relations. The propagated information will enable to us to evaluate predicates on different relations.

We call this technique of propagating information among different relations as *tuple ID propagation*. It propagates the IDs and class labels of target tuples to other relations. In the relation R to which the IDs and labels are propagated, each tuple t is associated with a set of IDs representing the target tuples joinable with t. In this way we can evaluate the classification power of each predicate in R, and useful predicates can be easily found. Tuple ID propagation is a convenient and flexible method for virtually joining different relations, with as low cost as possible. Since IDs can be easily propagated between any two relations, one can search freely in multiple relations via any join path, just by continuously propagating IDs. No repeated computation is needed when searching along different join paths that share common prefixes.

Based on the above method, we propose CrossMine, a scalable and accurate approach for multi-relational classification. The main idea of CrossMine is to repeatedly divide the target relation into partitions, and recursively work on each partition. There are two different algorithms of CrossMine: CrossMine-Tree and CrossMine-Rule. CrossMine-Tree is a decision-tree based classifier, which recursively selects the best attribute and divides all target tuples into partitions. CrossMine-Rule is a rule-based classifier, which repeatedly builds predictive rules and then focuses on remaining target tuples.

The remaining of this chapter is organized as follows. The problem definition is presented in Section 3.2. Section 3.3 introduces the idea of tuple ID propagation and its theoretical background. We describe the algorithm and implementation issues in Section 3.4. Experimental results are presented in Section 3.5. We discuss related work in Section 3.6 and how to adapt CrossMine to data stored on disks in Section 3.7.

3.2 Problem Definitions

In this section we will introduce the basic concepts of rule-based and decision-tree-based multi-relational classification. We will start from the basic definitions of predicates and rules, and then briefly explain the procedure of two classification algorithms.

3.2.1 Predicates

A predicate is the basic element of a rule. It is a constraint on a certain attribute in a certain relation, and a target tuple either satisfies it or not. For example, predicate "$p_1 = Loan(L, \_ , \_ , \_ , >= 12, \_)$" means that the duration of loan L is no less than 12 months. A predicate is often defined based on a certain join path. For example, "$p_2 = Loan(L, A, \_ , \_ , \_ , \_),\ Account(A,\ \_ , monthly, \_)$" is defined on the join path $Loan \bowtie Account$, which means that the associated account of a loan has frequency "*monthly*".

There are three types of predicates:

1. **Categorical predicate**: A categorical predicate a constraint that a categorical attribute must take a certain value, such as p_2 in the above example.

2. **Numerical predicate**: A numerical predicate is defined on a numerical attribute. It contains a certain value and a comparison operator, such as p_1 in the above example.

3. **Aggregation predicate**: An aggregation predicate also defined on numerical attributes. It is a constraint on the aggregated value of an attribute. It contains an aggregation operator, a certain value, and

a comparison operator. For example, $p_3 = Loan(L, A, \_, \_, \_, \_)$, $Order(\_, A, \_, \_, sum(amount) >= 1000, \_)$ is an aggregation predicate, which requires the sum of amount of all orders related to a loan is no less than 1000. The following aggregation operators can be used: *count*, *sum*, and *avg*.

Both CrossMine-Rule and CrossMine-Tree use tuple ID propagation (described in Section 3.3) to transfer information across different relations when building classifiers. To better integrate the information of ID propagation, they use *complex predicates* as elements of classifiers. A complex predicate $\hat{p}$ contains two parts:

1. *prop-path*: indicates how to propagate IDs. For example, the path "$Loan.account\_id \rightarrow Account.account\_id$" indicates propagating IDs from *Loan* relation to *Account* relation using *account_id*. *prop-path* could be empty if no ID propagation is involved.

2. *constraint*: indicates the constraint on the relation where the IDs are propagated to. It is actually a predicate that is either categorical, numerical, or involving aggregation.

A complex predicate is usually equivalent to two conventional predicates. For example, the rule "$Loan(L, +)$:– $Loan(L, A, \_, \_, \_, \_)$, $Account(A, \_, monthly, \_)$" can be represented by "$Loan(+)$:– $[Loan.account\_id \rightarrow Account.account\_id$, $Account.frequency = monthly]$".

3.2.2 Rules

A rule-based classification algorithm (such as CrossMine-Rule) aims at building a set of rules that can distinguish positive examples from negative ones. Each rule is a list of predicates, associated with a class label. A target tuple satisfies a rule if and only if it satisfies every predicate of the rule. In the example shown in Figure 3.1 where *Loan* is the target relation, there exists the following rule r:

$Loan(+)$:– $[Loan.account\_id \rightarrow Account.account\_id, Account.frequency = monthly]$.

We say a tuple t in *Loan* satisfies r if and only if **any** tuple in *Account* that is joinable with t has value "monthly" in the attribute of *frequency*. In this example, there are two tuples (with account-id 124 and 45) in *Account* satisfying the predicate "$Account(A, \_, monthly, \_)$". So there are four tuples (with loan-id 1, 2, 4, and 5) in *Loan* satisfying this rule.

In order to build rules, CrossMine-Rule needs to evaluate the prediction capability of predicates. There are different measures for this purpose, including information gain, gini-index [56], and foil gain [96]. Among these measures foil gain is most suitable for evaluating predicates, because it measures the change in entropy caused by each predicate. Thus we choose Foil gain as our measure.

Loan					
loan-id	account-id	amount	duration	payment	*class*
1	124	1000	12	120	+
2	124	4000	12	350	+
3	108	10000	24	500	−
4	45	12000	36	400	−
5	45	2000	24	90	+

Account		
account-id	frequency	date
124	monthly	960227
108	weekly	950923
45	monthly	941209
67	weekly	950101

Figure 3.2: An example database (the last column of Loan relation contains class labels.)

Definition 1 (Foil gain). *For a rule r, we use $P(r)$ and $N(r)$ to denote the number of positive and negative examples satisfying r. Suppose the current rule is r. We use $r + p$ to denote the rule constructed by appending predicate p to r. The foil gain of predicate p is defined as follows,*

$$I(r) = -\log \frac{P(r)}{P(r) + N(r)} \tag{3.1}$$

$$foil\_gain(p) = P(r + p) \cdot [I(r) - I(r + p)] \tag{3.2}$$

Intuitively, $foil\_gain(p)$ represents the total number of bits saved in representing positive examples by appending p to the current rule. It indicates how much the predictive power of the rule can be increased by appending p to it.

In order to build a rule, we repeatedly search for the predicate with highest foil gain and attach it to the current rule, until no gainful predicates can be found. This procedure will be described in Section 3.4.2.

3.2.3 Decision Trees

CrossMine-Tree is a decision-tree-based classifier. Each tree node n contains two parts: (1) *prop-path*, which indicates how tuple IDs are propagated, and (2) a splitter that divides all target tuples on n into several partitions, each corresponding to one of its child nodes. There are three types of tree nodes:

1. **Categorical node**: A categorical node is defined on a categorical attribute A. It has a child node for each value of A, and divides tuples according to their values on A.

2. **Numerical node**: A numerical node is defined on a numerical attribute. It is like a numerical predicate and has only two child nodes.

3. **Aggregation node**: An aggregation node is defined on the aggregated value of an attribute. It is similar to an aggregation predicate, and has only two child nodes.

A tree node can be considered as a list of predicates on the same attribute, each corresponding to a child node. Unlike decision trees for single tables, a target tuple t may be joinable with multiple tuples in other relations, thus t may satisfy multiple predicates in a tree node. Since t can only be assigned to one child node, the child nodes of a node should be ranked, and the first child node satisfied by t is chosen for t. In Section 3.4 we will describe how to determine the order of child nodes. CrossMine-Tree uses Information gain [95] to evaluate the classification capability of attributes, which is defined as follows.

Definition 2 (Information gain). *Suppose there are P positive and N negative examples at a tree node. Suppose an attribute A divides those examples into k parts, each containing P_i positive and N_i negative examples.*

$$entropy(P,N) = -\left(\frac{P}{P+N}\cdot\log\frac{P}{P+N}+\frac{N}{P+N}\cdot\log\frac{N}{P+N}\right) \tag{3.3}$$

$$info\_gain(A) = entropy(P,N) - \sum_{i=1}^{k}\frac{P_i+N_i}{P+N}\cdot entropy(P_i,N_i) \tag{3.4}$$

The main idea of decision tree algorithm is to find the attribute with highest information gain, partition all target tuples using that attribute, and then work on each partition recursively. The CrossMine-Tree algorithm will be described in Section 3.4.1.

3.3 Tuple ID Propagation

In this section we present the method of tuple ID propagation. In general, tuple ID propagation is a method for transferring information among different relations by virtually joining them. It is a convenient method that enables flexible search in relational databases, and is much less costly than physical join in both time and space.

3.3.1 Searching for Predicates by Joins

Consider the mini example database in Figure 3.2, together with two other relations shown in Figure 3.3. Suppose we want to compute the foil gain of predicates in all non-target relations (*Account*, *Order*, and *Transaction*). A possible approach is to join all four relations into one as in Figure 3.4 (joined by account-id), and compute the foil gain of each predicates. However, the foil gain computed based on the joined relation may be highly biased, since different target tuples appear different times in the relation. For example, predicate "*Loan.duration* = 24" is satisfied by six negative and two positive tuples in the joined relation, and should have high foil gain accordingly. But in fact it is only satisfied by one positive and one negative tuple in the *Loan* relation, and thus has low foil gain. This problem cannot be solved by assigning weights to tuples in the joined relation. Suppose we set the weight of each tuple in the joined relation to one over the number of times the corresponding target tuple appears, which is a most reasonable way to assign weights because it guarantees the correct statistics for predicates satisfied by all tuples. Then predict "$Transaction.operation$ = withdrawal" is satisfied by 3 positive and 1.33 negative tuples. But it is actually satisfied by 3 positive and 2 negative tuples in *Loan* relation (as defined in Section 3.2.2).

Moreover, joining many relations together often leads to unaffordable cost in computation and storage. In this mini example where four relations are joined directly and each $account-id$ only appears a few times, the joined relation is still much larger than the original relations. In the real database where each relation has at least thousands of tuples and each $account-id$ appears tens or hundreds of times in relations such as *Transaction* or *Order*, if we join all original relations together, there can be millions or billions of tuples in the joined relation, which is unaffordable to store or to compute.

Order			
order-id	account-id	amount	type
1	45	500	misc
2	45	800	house
3	108	1000	car
4	108	2000	house
5	124	5000	car
6	124	400	misc

Transaction			
trans-id	account-id	operation	amount
1	45	withdrawal	100
2	108	deposit	800
3	108	withdrawal	400
4	108	deposit	100
5	124	withdrawal	200

Figure 3.3: *Order* and *Transaction* relation in the mini example database

Loan ⋈ Account ⋈ Order ⋈ Transaction											
l-id	a-id	l-amnt	dur	pay	freq	date	o-amnt	o-type	oper	t-amnt	*class*
1	124	1000	12	120	monthly	960227	5000	car	withdrawal	200	+
1	124	1000	12	120	monthly	960227	400	misc	withdrawal	200	+
2	124	4000	12	350	monthly	960227	5000	car	withdrawal	200	+
2	124	4000	12	350	monthly	960227	400	misc	withdrawal	200	+
3	108	10000	24	500	weekly	950923	1000	car	deposit	800	−
3	108	10000	24	500	weekly	950923	1000	car	withdrawal	400	−
3	108	10000	24	500	weekly	950923	1000	car	deposit	100	−
3	108	10000	24	500	weekly	950923	2000	house	deposit	800	−
3	108	10000	24	500	weekly	950923	2000	house	withdrawal	400	−
3	108	10000	24	500	weekly	950923	2000	house	deposit	100	−
4	45	12000	36	400	monthly	941209	500	misc	withdrawal	100	−
4	45	12000	36	400	monthly	941209	800	house	withdrawal	100	−
5	45	2000	24	90	monthly	941209	500	misc	withdrawal	100	+
5	45	2000	24	90	monthly	941209	800	house	withdrawal	100	+

Figure 3.4: The joined relation of *Loan*, *Account*, *Order*, and *Transaction*

In order to avoid creating huge joined relations, some ILP approaches (e.g., FOIL [96]) repeatedly join different relations along different join paths. For example, the target relation *Loan* may be joined with *Account*, *Order*, *Transaction* and *Disposition* separately, and then each joined relation is further joined with other relations (e.g., *Card*, *Client*, and *District*) to explore information in them. Because each relation is usually joinable with several relations, one needs to explore a large number of different join paths during the process of building a classifier, and needs to perform a physical join for each join path explored. This is very expensive since the joined relation is often much larger than the original relations (as shown in Figure 3.4). In the next section we will introduce *tuple ID propagation*, a technique for performing virtual joins among relations, which is much less expensive than physical joins. When searching for good predicates, one can propagate tuple IDs between any two relations, which requires much less computation and storage cost compared with creating joined relations. This makes it possible to "navigate freely" among different relations when building classifiers.

3.3.2 Tuple ID Propagation

For simplicity we assume the primary key of the target relation is an attribute of integers, which represents the ID of each target tuple (we can create such a primary key if there is not one). Consider the example database shown in Figure 3.5, which has the same schema as in Figure 3.2. Instead of performing physical join, the IDs and class labels of target tuples can be propagated to *Account* relation. The procedure is formally defined as follows.

Definition 3 (Tuple ID propagation). *Suppose two relations R_1 and R_2 can be joined by attributes $R_1.A$ and $R_2.A$. Each tuple t in R_1 is associated with a set of IDs in the target relation, represented by $idset(t)$. For each tuple u in R_2, we set $idset(u) = \bigcup_{t \in R_1, t.A=u.A} idset(t)$.*

The following lemma and its corollary show the correctness of tuple ID propagation and how to compute foil gain from the propagated IDs.

Lemma 1 *Suppose two relations R_1 and R_2 can be joined by attributes $R_1.A$ and $R_2.A$, and R_1 is the target relation with primary key $R_1.id$. With tuple ID propagation from R_1 to R_2 via join $R_1.A = R_2.A$, for each tuple u in R_2, $idset(u)$ represents all target tuples joinable with u via join $R_1.A = R_2.A$.*

Proof. *From definition 3, we have $idset(u) = \bigcup_{t \in R_1, t.A=u.A} idset(t)$. That is, $idset(u)$ represents the target tuples joinable with u using join $R_1.A = R_2.A$.*

Loan			
loan-id	account-id	···	*class*
1	124		+
2	124		+
3	108		−
4	45		−
5	45		+

Account				
account-id	frequency	date	IDs	*class labels*
124	monthly	960227	1, 2	2+, 0−
108	weekly	950923	3	0+, 1−
45	monthly	941209	4, 5	1+, 1−
67	weekly	950101	–	0+, 0−

Figure 3.5: Example of tuple ID propagation (some attributes in *Loan* are not shown).

Corollary 1 *Suppose two relations R_1 and R_2 can be joined by attribute $R_1.A$ and $R_2.A$, R_1 is the target relation, and all tuples in R_1 satisfy the current rule (others have been eliminated). If R_1's IDs are propagated to R_2, then the foil gain of every predicate in R_2 can be computed using the propagated IDs on R_2.*

Proof. *Given the current rule r, for a predicate p in R_2, such as $R_2.B = b$, its foil gain can be computed based on $P(r)$, $N(r)$, $P(r + p)$ and $N(r + p)$. $P(r)$ and $N(r)$ have been computed during the process of building the current rule. $P(r+p)$ and $N(r+p)$ can be calculated by finding all target tuples that are joinable with any tuple t in R_2 that satisfies predicate p.*

For example, suppose "$Loan(L, +)$:– $Loan(L, A, \_\,, \_\,, \_\,, \_\,)$" is the current rule. For predicate "$Account(A, \_\,, monthly, \_\,)$", we first find out tuples in *Account* relation that satisfy this predicate, which are {124, 45}. Then we find out tuples in *Loan* relation that can be joined with these two tuples ({1, 2, 4, 5}), which contain three positive and one negative examples. Then we can easily compute the foil gain of predicate "$Account(A, \_\,, monthly, \_\,)$".

Besides propagating IDs from the target relation to relations directly joinable with it, one can also propagate IDs transitively by propagating the IDs from one non-target relation to another, according to the following lemma.

Lemma 2 *Suppose two non-target relations R_2 and R_3 can be joined by attribute $R_2.A$ and $R_3.A$. For each tuple v in R_2, $idset(v)$ represents the target tuples joinable with v (using the join path specified by the current rule). By propagating IDs from R_2 to R_3 through the join $R_2.A = R_3.A$, for each tuple u in R_3, $idset(u)$ represents target tuples that can be joined with u (using the join path in the current rule, plus the join $R_2.A = R_3.A$).*

Proof. *Suppose a tuple u in R_3 can be joined with v_1, v_2, $\cdots$, v_m in R_2, using join $R_2.A = R_3.A$. Then $idset(u) = \bigcup_{i=1}^{m} idset(v_i)$. A target tuple t is joinable with any one of v_1, v_2, $\cdots$, v_m if and only if $t.id \in \bigcup_{i=1}^{m} idset(v_i)$. Therefore, a target tuple t is joinable with u (using the join path in the current rule, plus the join $R_2.A = R_3.A$) if and only if $t.id \in idset(u)$.*

A corollary similar to Corollary 1 can be proved for Lemma 2. That is, by tuple ID propagation between different relations, one can also compute the foil gain of predicates based on the propagated IDs on each relation.

3.3.3 Analysis and Constraints

Tuple ID propagation is a method to perform virtual join. Instead of physically joining relations, they are virtually joined by attaching the IDs of target tuples to tuples in non-target relations. In this way the predicates can be evaluated as if physical join is performed. Tuple ID propagation is an efficient method, since IDs can be easily propagated from one relation to another, with small amounts of data transfer and extra storage space. By doing so, predicates in different relations can be evaluated with little redundant computation.

Consider the example in Figure 3.2 and 3.3. These four relations contain 86 entries in total. If we join all four tables as in Figure 3.4, a joined relation of 168 entries is created. If we propagate IDs from *Loan* relation to the other three relations, we only need 22 entries for propagated IDs. Even in this mini example where only direct joins are explored, tuple ID propagation can save much space. The savings in storage and computational cost will be much more significant on real databases with large relations and if longer join paths are explored.

Tuple ID propagation is also a highly flexible method. One can associate a relation with multiple sets of IDs corresponding to different join paths, and use any set for further propagation. Compared to previous approaches [6, 15], tuple IDs can be propagated freely between any two relations, following any join paths. These features enables CrossMine to search for predicates freely and efficiently in databases with complex schemas.

ID propagation, though valuable, should be enforced with certain constraints. There are two cases that such propagation could be counter-productive: (1) propagate via large fan-outs, and (2) propagate via long weak links. The first case happens if after IDs are propagated to a relation R, it is found that every tuple in R is joined with many target tuples and every target tuple is joined with many tuples in R. Then the semantic link between R and the target relation is usually very weak because the link is very unselective. For example, propagation among people via birth-country links may not be productive. The second case happens if the propagation goes through very long links, e.g., linking a student with his teacher's group mate's students may not be productive, either. From the consideration of efficiency and accuracy, CrossMine discourages propagation via such links.

3.4 Building Classifiers

In this section we present the algorithms of CrossMine for building classifiers. In general, CrossMine is a divide-and-conquer algorithm, which searches for the best way to split the target relation into partitions, and then recursively works on each partition. CrossMine uses tuple ID propagation to efficiently search for good predicates or tree nodes among different relations, in order to build accurate classifiers based on relational information.

CrossMine-Tree and CrossMine-Rule use different methods to split the target relation. CrossMine-Tree searches for the best tree node in each step, and split the target tuples according to the attribute of the tree node. In this way it recursively builds a decision tree for classification. In contrast, CrossMine-Rule generates a good rule in each step, and splits the target tuples according to whether they satisfy the rule. Then it focuses on the tuples not satisfying the rule to generate more rules. CrossMine-Tree is usually more efficient because it usually splits the target tuples into small partitions, while CrossMine-Rule is usually more accurate because each rule is built based on a large set of tuples. We will first introduce the overall procedures of CrossMine-Tree and CrossMine-Rule, then describe the detailed methods for searching for predicates or attributes by tuple ID propagation in Section 3.4.3.

3.4.1 The CrossMine-Tree Algorithm

CrossMine-Tree is a divide-and-conquer algorithm that builds a decision tree for a relational database. It starts from all the target tuples, and recursively finds the best attribute to divide the tuples into partitions. It stops working on one branch if there is not enough tuples or no attribute with sufficient information gain can be found.

Because of the huge number of join paths starting from the target relation, it is impossible to perform an exhaustive search by evaluating all possible attributes and selecting the best one. On the other hand, most ILP approaches can successfully confine the searching process in promising directions by keep searching the neighborhood of relations that are used in the current rule or decision tree. Therefore, CrossMine-Tree starts from the neighborhood of the target relation, and gradually expands the search range, as described below.

Initially, CrossMine-Tree works on all target tuples and set the target relation to *active*. At each step, it searches for the best attribute A in all active relations and relations joinable with any active relation, and uses A to divide the target tuples. The relation containing A is also set to *active*. Then CrossMine-Tree works on each partition of tuples, and recursively builds a tree for that partition. A tree node is not further divided if the number of target tuples is less than MIN_SUP, or the maximum *info_gain* of any attribute is less than MIN_INFO_GAIN. The algorithm is shown in Figure 3.6.

3.4.2 The CrossMine-Rule Algorithm

Unlike CrossMine-Tree, CrossMine-Rule builds a classifier containing a set of rules, each containing a list of complex predicates and a class label. The overall procedure is to build rules one by one. After a rule r is built, all positive target tuples satisfying r are removed from the dataset. The negative tuples are never removed, so each rule is built with all remaining positive tuples and all negative tuples, which guarantees

Algorithm 1 Build-Tree

Input: A relational database D with a target relation R_t.

Output: A decision tree for predicting class labels of target tuples.

```
Procedure
    tree node $n \leftarrow$ empty-node
    if $|R_t| <$ MIN_SUP then
        return
    evaluate all attributes in any active relation or relations joinable with active relation
    $A \leftarrow$ attribute with max information gain
    if $info\_gain(A) <$ MIN_INFO_GAIN then
        return
    $n.attr \leftarrow A$
    set relation of $A$ to active
    divide $R_t$ according to $A$
    for each partition $R_t'$
        tree node $n_i \leftarrow$ Build-Tree($D$,$R_t'$)
        add $n_i$ as a child of $n$
    end for
    for each relation $R'$ that is set active by this function
        set $R'$ inactive
    end for
    return $n$
```

Figure 3.6: Algorithm *Build-Tree*

the quality of the rules. The algorithm is shown in Figure 3.7.

To build a rule, CrossMine-Rule repeatedly searches for the best complex predicate and appends it to the current rule, until the no gainful predicate can be found. This can be considered as a general-to-specific learning procedure. We start from the most general rule with no predicates (and thus is satisfied by any target tuple), keep adding predicates to make it more and more specific, until we cannot further improve it.

Because of the huge hypothesis space, we also need to confine the rule-building process in promising directions instead of searching aimlessly. We say a relation is *active* if it appears in the current rule. Every active relation is required to have the correct propagated IDs on every tuple before searching for the next best predicate. When searching for a predicate, all possible predicates on any active relation or relation joinable with any active relation will be evaluated. The algorithm for searching for best predicate will be described in Section 3.4.3. When there are more than two classes of target tuples, CrossMine-Rule builds a classifier for each class.

3.4.3 Finding Best Predicate or Attribute

CrossMine needs to search for good predicates (or attributes) in different relations, to which tuple IDs have been propagated to. Suppose the best predicate/attribute needs to be found in a certain relation R.

Algorithm 2 Find-A-Rule

Input: A relational database D with a target relation R_t.

Output: A set of rules for predicting class labels of target tuples.

```
Procedure
    rule set $R \leftarrow \emptyset$
    while(true)
        rule $r \leftarrow$ empty-rule
        set $R_t$ to active
        do
            Complex predicate $p \leftarrow$ the predicate with highest foil gain
            if $foil\_gain(p) <$ MIN_FOIL_GAIN
            then break
            else
                    $r \leftarrow r + p$
                    remove all target tuples not satisfying $r$
                    update IDs on every active relation
                    if $p.constraint$ is on an inactive relation
                    then set that relation active
            end else
        while($r.length <$ MAX_RULE_LENGTH)
        if $r =$ empty-rule then break
        $R \leftarrow R \cup \{r\}$
        set all relations inactive
    end while
    return $R$
```

Figure 3.7: Algorithm *Find-A-Rule*

CrossMine evaluates all possible predicates (or attributes) in R and selects the best one. A method similar to RainForest [48] is used to efficiently find out the numbers of tuples of different classes that have each value on each attribute.

Suppose CrossMine-Rule is searching for the best predicate on a categorical attribute A_c. For each value a_i of A_c, a predicate $p_i = [R.A_c = a_i]$ is built. CrossMine-Rule scans the values of each tuple on A_c to find out the numbers of positive and negative target tuples satisfying each predicate p_i. Then the foil gain of each p_i can be computed and the best predicate can be found.

As mentioned in Section 3.2.3, a target tuple t may join with multiple tuples in other relations, thus the child nodes of a tree node should be ranked, since t can only be assigned to one child node. A greedy approach is used to order the nodes. CrossMine-Tree first selects the child node with highest information gain, then the second one, and so on. Finally a child node is built for those target tuples that are not assigned to all other nodes.

To find the best predicate on a numerical attribute A_n, suppose a sorted index for A_n has been built beforehand. CrossMine-Rule (or CrossMine-Tree) iterates from the smallest value of A_n to the largest one. A pool of tuple IDs is maintained, so that when iterating to each value v_i, all target tuples satisfying predicate $[A_n \leq v_i]$ or $[A_n > v_i]$ can be found. In this way, one can compute the foil gain (or information gain) of using each value to partition the tuples. To search for the best aggregation predicate for A_n, CrossMine-Rule (or CrossMine-Tree) first computes the corresponding statistics (count, sum, and average) for each target tuple, by scanning the IDs associated with tuples in R. Then the foil gain (or information gain) of all aggregation predicates (or tree nodes) can be computed using a method similar to that for finding the best numerical predicates.

In each step of CrossMine the best predicate or attribute is searched in all the active relations, or relations joinable with any active relation by propagating IDs. Consider the database in Figure 3.1. At first only *Loan* relation is active. Suppose the first best predicate is in *Account* relation, which becomes active as well, and CrossMine will try to propagate the tuple IDs from *Loan* or *Account* to other relations to find the next best predicate. In this way the search range can be gradually expanded along promising directions, which avoid aimless search in the huge hypothesis space.

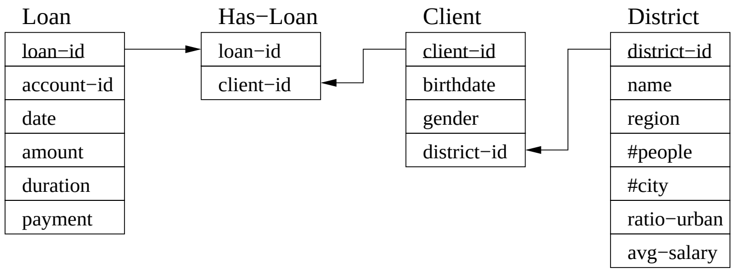


Figure 3.8: Another example database

The above algorithm may fail to find good predicates in databases containing some relations that are used to join with other relations (Figure 3.8). There is no useful attribute in *Has_Loan* relation, thus the algorithm will never use information out of *Loan* relation. CrossMine uses *look-one-ahead* method [13] to solve this problem. After IDs have been propagated to a relation $\bar{R}$, if $\bar{R}$ contains a foreign-key pointing to relation $\bar{R}'$, IDs are propagated from $\bar{R}$ to $\bar{R}'$, and used to search for good predicates (or attributes) in $\bar{R}'$. In this way, information in *Client* relation and *District* relation can be used in building classifiers.

To achieve high accuracy in multi-relational classification, an algorithm should be able to find most of the useful predicates (or attributes) in the database and build good classifiers with them. There are two types of relations in most commercial databases following the E-R model design: *entity* relation and *relationship* relation. Usually each entity relation is reachable from some other entity relations via join paths going through relationship relations. With the usage of *prop-path* in predicates or attributes, and the method of *look-one-ahead*, CrossMine can search across two joins at a time. It can also gradually expand the search range based on relations containing useful predicates or attributes. This enables CrossMine to build accurate classifiers while confining the search process in promising directions.

3.4.4 Predicting Class Labels

After generating a classifier, one needs to predict the class labels of unlabeled target tuples. Suppose a rule $r = R_t(+) \coloneq p_1, p_2, \ldots, p_k$. (where each p_i is a complex predicate.) CrossMine-Rule propagates the IDs of all target tuples along the prop-path of each predicate p_i, and prune all IDs of target tuples not satisfying the constraint of p_i. For each target tuple t, the most accurate rule that is satisfied by t is found, and the class label of that rule is used as the predicted class. Similar to CrossMine-Rule, CrossMine-Tree recursively propagates tuple IDs from the root of the tree to the leaf nodes. Each target tuple belongs to only one leaf node, and the class label of that node is used as the prediction.

3.4.5 Tuple Sampling

From the algorithm of CrossMine-Rule we can see that during the procedure of building rules, the number of positive tuples keeps decreasing and the number of negative tuples remains unchanged. Let $r.sup^+$ and $r.sup^-$ be the number of positive and negative tuples satisfying a rule r. Let $r.bg^+$ and $r.bg^-$ be the number of positive and negative tuples from which r is built. We estimate the accuracy of r by $Accuracy(r) = (r.sup^+ + 1)/(r.sup^+ + r.sup^- + c)$ [24]. Usually the first several rules can cover the majority of the positive tuples. However, it still takes a similar amount of time to build a rule because $r.bg^-$ is large. When $r.bg^+$ is small, even if $r.bg^-$ is large, the quality of r cannot be guaranteed. That is, one cannot be confident that $Accuracy(r)$ is a good estimate for the real world accuracy of r. Therefore, although much time is spent in building these rules, the quality of them is usually much lower than that of the rules with high bg^+ and bg^-.

Based on this observation, the following method is proposed to improve the efficiency of CrossMine-Rule. Before a rule is built, we require that the number of negative tuples is no greater than NEG_POS_RATIO times the number of positive tuples. Sampling is performed on the negative tuples if this requirement is not satisfied. We also require that the number of negative tuples is smaller than MAX_NUM_NEGATIVE, which is a large constant.

When sampling is used, the accuracy of rules should be estimated in a different way. Suppose before building rule r, there are P positive and N negative tuples. N' negative tuples are randomly chosen by sampling ($N' < N$). After building rule r, suppose there are p positive and n' negative tuples satisfying r. We need to estimate n, the number of negative tuples satisfying r. The simplest estimation is $n \approx n' \frac{N}{N'}$. However, this is not a safe estimation because it is quite possible that r luckily excludes most of the N' negative examples but not the others. We want to find out a number n, so that the probability that $n' \leq n \frac{N'}{N}$

is 0.9. Or to say, it is unlikely that $\frac{n}{N} \leq \frac{n'}{N'}$.

As we know, N' out of N negative tuples are chosen by sampling. Assume we already know that n negative tuples satisfy r. Consider the event of a negative tuple satisfying r as a random event. Then n' is a random variable obeying binomial distribution, $n' \sim B(N', \frac{n}{N})$. n' can be considered as the sum of N' random variable of $B(1, \frac{n}{N})$. When N' is large, according to central limit theorem, we have $\frac{n'}{N'} \sim N(\frac{n}{N}, \frac{\frac{n}{N}(1-\frac{n}{N})}{N'})$. For a random variable $X \sim N(\mu, \sigma^2)$, $P(X \geq \mu - 1.28\sigma) \approx 0.9$. So we require

$$\frac{n'}{N'} = \frac{n}{N} - 1.28\sqrt{\frac{\frac{n}{N}(1-\frac{n}{N})}{N'}} \tag{3.5}$$

Let $x = \frac{n}{N}$ and $d = \frac{n'}{N'}$. Equation (3.5) is converted into

$$\left(1 + \frac{1.64}{N'}\right)x^2 - \left(2d + \frac{1.64}{N'}\right)x + d^2 = 0 \tag{3.6}$$

Equation (3.6) can be easily solved with two solutions x_1 and x_2, corresponding to the positive and negative squared root in equation (3.5). The greater solution x_2 should be chosen because it corresponds to the positive squared root. If there are x_2N negative tuples satisfying the rule before sampling, then it is unlikely that there are less than n' tuples satisfying the rule after sampling. Therefore, we use x_2N as the safe estimation of n, by which we can estimate the accuracy of r.

3.5 Experimental Results

We performed comprehensive experiments on both synthetic and real databases to show the accuracy and scalability of CrossMine-Rule and CrossMine-Tree. They are compared with FOIL [96] and TILDE [13] in every experiment. The source code of FOIL and binary code of TILDE are from their authors. CrossMine-Rule, CrossMine-Tree and FOIL ran on a 1.7GHz Pentium 4 PC with Windows 2000. TILDE ran on a Sun Blade 1000 workstation. Ten-fold experiments are used unless specified otherwise. The following parameters are used in CrossMine-Rule. MIN_FOIL_GAIN = 2.5. MAX_RULE_LENGTH = 6, NEG_POS_RATIO = 1, MAX_NUM_NEGATIVE = 600. The following are used in CrossMine-Tree. MIN_INFO_GAIN = 0.05, MIN_SUP = 10. The performances of CrossMine-Rule and CrossMine-Tree are not sensitive to above parameters according to our experiences.

3.5.1 Synthetic Databases

To evaluate the scalability of CrossMine-Rule and CrossMine-Tree, a set of synthetic relational databases are generated, which mimic the real relational databases. Our data generator takes the parameters in Table 3.1. We first generate a relational schema with $|R|$ relations, one being the target relation. The number of attributes of each relation obeys exponential distribution with expectation A and is at least A_{min}. One of the attributes is the primary-key. All attributes are categorical, and the number of values of each attribute (except primary key) obeys exponential distribution with expectation V and is at least V_{min}. Besides these attributes, each relation has a few foreign-keys, pointing to primary-keys of other relations. The number of foreign-keys of each relation obeys exponential distribution with expectation F and is at least F_{min}.

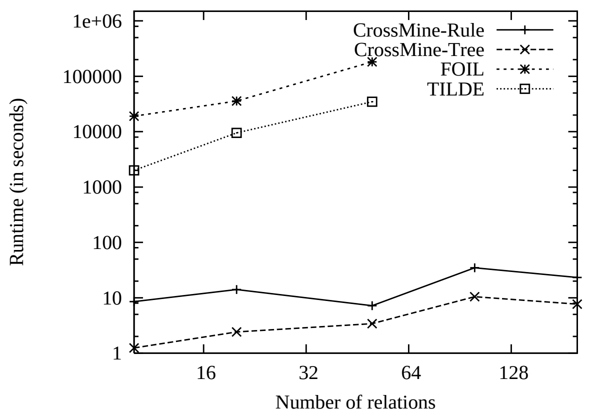
Name	Description	Default value
$\|R\|$	# relations	x
T_{min}	Min # tuples in each relation	50
T	Expected # tuples in each relation	y
A_{min}	Min # attributes in each relation	2
A	Expected # attributes in each relation	5
V_{min}	Min # values of each attribute	2
V	Expected # values of each attribute	10
F_{min}	Min # foreign-keys in each relation	2
F	Expected # foreign-keys in each relation	z
$\|r\|$	# rules	10
L_{min}	Min # complex predicates in each rule	2
L_{max}	Max # complex predicates in each rule	6
f_A	Prob. of a predicate on active relation	0.25

Table 3.1: Parameters of data generator.

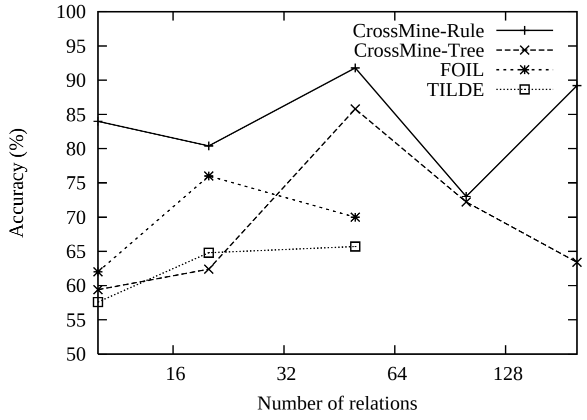
After the schema is generated, we generate rules that are lists of complex predicates, with randomly generated class labels. The number of complex predicates in each rule obeys uniform distribution between L_{min} and L_{max}. Each predicate has probability f_A to be on an active relation and probability $(1 - f_A)$ to be on an inactive relation (involving a propagation).

Then we generate tuples. The target relation has exactly T tuples. Each target tuple is generated according to a randomly chosen rule. In this way we also need to add tuples to non-target relations to satisfy the rule. After generating target tuples, we add more tuples to non-target relations. The number of tuples in each relation R obeys exponential distribution with expectation T and is at least T_{min}. We use "$Rx.Ty.Fz$" to represent a database with x relations, expected y tuples in each relation, and expected z foreign-keys in each relation.

To test the scalability and accuracy of the approaches, we perform experiments on databases with different numbers of relations, different numbers of tuples in each relation, and different numbers of foreign-keys in



a) Time vs. # of relations

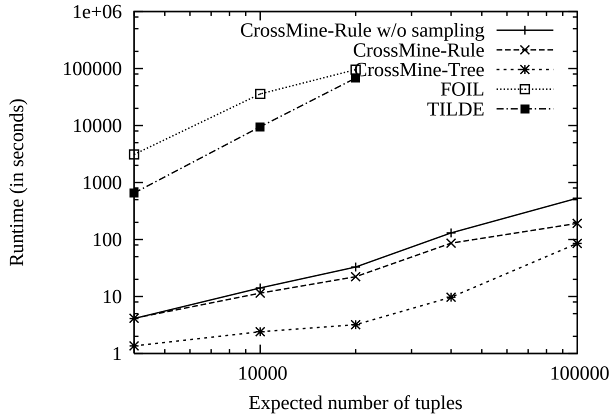


b) Accuracy vs. # of relations

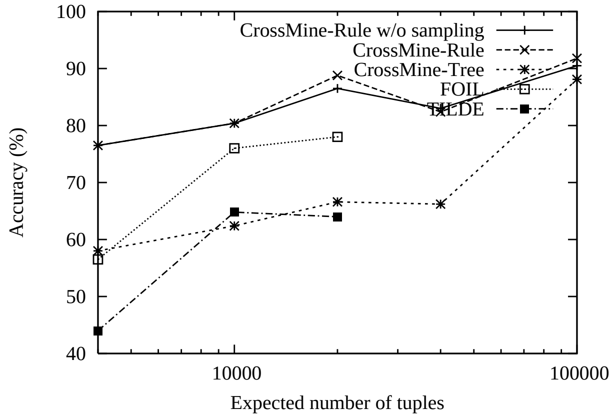
Figure 3.9: Runtime and accuracy on R*.T500.F2.

each relation. The running time and accuracy of CrossMine-Rule, CrossMine-Tree FOIL, and TILDE are compared.

To test the scalability w.r.t. the number of relations, five databases are created with 10, 20, 50, 100, and 200 relations respectively. In each relation the expected number of tuples is 500 and the expected number of foreign-keys is 2. Figure 3.9 (a) shows the running time of the four methods. Ten-fold experiments are used in most tests, and the average running time is reported. If the running time of an algorithm is greater than 10 hours, only one fold is used and further experiments are stopped. One can see that CrossMine-Rule and CrossMine-Tree are hundreds of times faster than FOIL and TILDE, and they are much more scalable. Their accuracies are shown in Figure 3.9 (b). One can see that CrossMine-Rule is more accurate than FOIL and TILDE. CrossMine-Tree achieves high accuracy on some datasets.



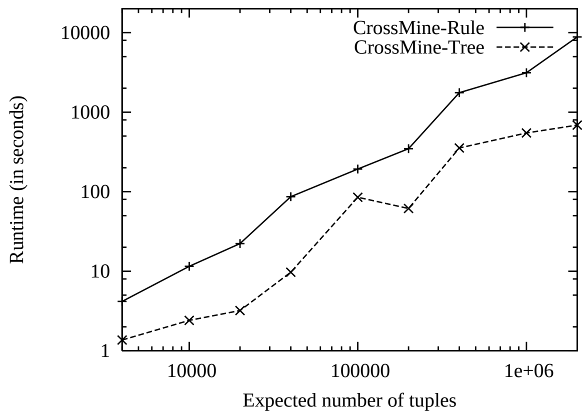
a) Time vs. # of tuples



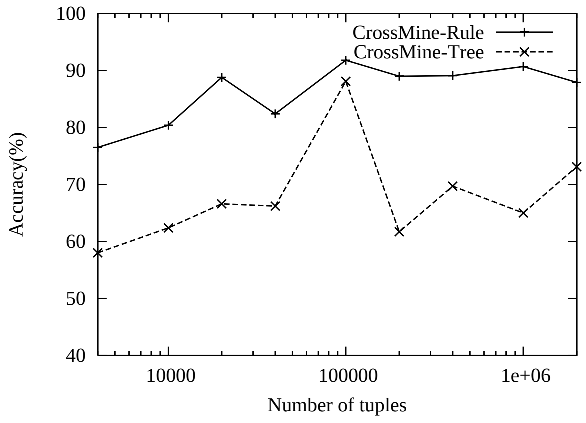
b) Accuracy vs. # of tuples

Figure 3.10: Runtime and accuracy on R20.T*.F2.

To test the scalability w.r.t. the number of tuples, five databases are created with the expected number of tuples in each relation being 200, 500, 1000, 2000, and 5000, respectively. There are twenty relations in each dataset, thus the expected number of tuples ranges from 4K to 100K. The expected number of foreign-keys in each relation is 2. In this experiment, the performances of CrossMine-Rule with and without sampling are tested to show the effectiveness of sampling method. Figure 3.10 (a) shows the running time of the four methods. One can see that CrossMine-Rule and CrossMine-Tree are much more scalable than FOIL and TILDE. When the number of tuples increases from 4K to 20K, the running time of CrossMine-Rule and CrossMine-Tree increase 8 times and 2.4 times, while those of FOIL and TILDE increase 30.6 times and 104 times. It is also shown that tuple sampling nontrivially improves the efficiency of CrossMine-Rule. The accuracy of the three methods is shown in Figure 3.10 (b). CrossMine-Rule is more accurate than other methods, and the sampling method only slightly affects the accuracy.



a) Time vs. # of tuples

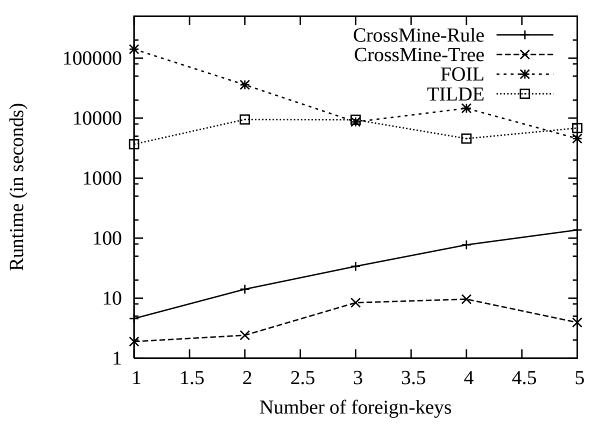


b) Accuracy vs. # of tuples

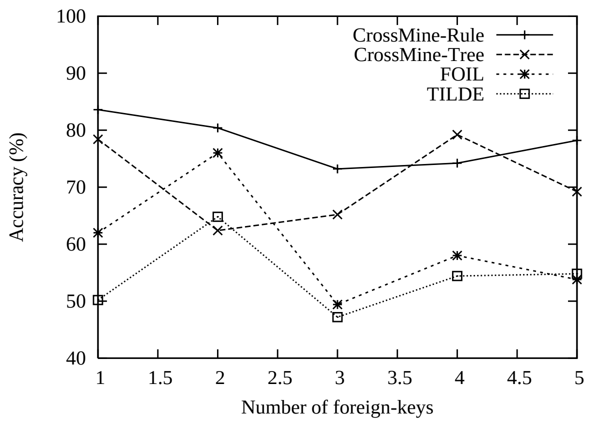
Figure 3.11: Scalability of CrossMine-Rule and CrossMine-Tree.

We also test the scalability of CrossMine-Rule and CrossMine-Tree on larger datasets with up to 2M tuples, as shown in Figure 3.11. It can be seen that both of them are highly scalable w.r.t. the number of tuples, and can run on very large databases. CrossMine-Tree is more efficient than CrossMine-Rule, while CrossMine-Rule is more accurate.

Finally we test the scalability w.r.t. the number of foreign-keys. Five databases are created with the expected number of foreign-keys in each relation being 1 to 5. The running time of four methods are shown in Figure 3.12 (a) and the accuracies shown in Figure 3.12 (b). One can see that the running time of FOIL and TILDE decreases as number of foreign-keys increase, because it is easier for them to find predicates. However, CrossMine-Rule and CrossMine-Tree are still much more efficient than FOIL and TILDE.



a) Time vs. # of foreign-keys



b) Accuracy vs. # of foreign-keys

Figure 3.12: Runtime and accuracy on R20.T500.F*.

3.5.2 Real Databases

Experiments are conducted on two real databases to compare the efficiency and accuracy of CrossMine-Rule, CrossMine-Tree, FOIL, and TILDE. The first database is the financial database used in PKDD CUP 1999, whose schema is shown in Figure 3.1. We modify the original database by shrinking 90% of *Trans* relation which was extremely huge, and randomly removing some positive tuples in *Loan* relation to make the numbers of positive tuples and negative tuples more balanced. The final database contains eight relations and 75982 tuples in total. The Loan relation contains 324 positive tuples and 76 negative ones. The performances on this database is shown in Table 3.2.

Approach	Accuracy	Runtime
CrossMine-Rule w/o sampling	89.5%	20.8 sec
CrossMine-Rule	88.3%	16.8 sec
CrossMine-Tree	87.3%	8.23 sec
FOIL	74.0%	3338 sec
TILDE	81.3%	2429 sec

Table 3.2: Performances on the financial database of PKDD CUP'99.

The second database is the Mutagenesis database, which is a frequently used ILP benchmark. It contains four relations and 15218 tuples. The target relation contains 188 tuples, with 124 being positive and 64 being negative. This dataset is pretty small and sampling method has no influence on CrossMine-Rule. The performance is shown in Table 3.3. The accuracy of CrossMine-Tree is quite low because the decision-tree based method is not quite suitable for small datasets. Since this data set contains only a very small number of relations, CrossMine-Rule does not achieve higher accuracy than FOIL and TILDE.

In general, one can see that CrossMine-Rule is a highly accurate and scalable approach. Although

Approach	Accuracy	Runtime
CrossMine-Rule	89.3%	2.57 sec
CrossMine-Tree	75.0%	0.66 sec
FOIL	79.7%	1.65 sec
TILDE	89.4%	25.6 sec

Table 3.3: Performances on the Mutagenesis database.

CrossMine-Tree is not as accurate as CrossMine-Rule, it is significantly more efficient. Thus we may use boosting technique to improve its accuracy.

3.6 Related Work

The most important category of approaches in multi-relational classification is ILP [80, 73], which is defined as follows. Given background knowledge B, a set of positive examples P, and a set of negative examples N, find a hypothesis H, which is a set of Horn clauses such that: (1) $\forall p \in P : H \cup B \models p$ (completeness), and (2) $\forall n \in N : H \cup B \not\models n$ (consistency).

The well known ILP systems include FOIL [96], Golem [82], and Progol [81]. FOIL is a top-down learner, which builds clauses that cover many positive examples and few negative ones. Golem is a bottom-up learner, which performs generalizations from the most specific clauses. Progol uses a combined search strategy. Some recent approaches TILDE [13], Mr-SMOTI [5], and RPTs [83] use the idea of C4.5 [95] and inductively construct decision trees from relational data.

Most ILP approaches do not have high scalability w.r.t. numbers of relations and numbers of tuples. An ILP approach needs to evaluate features in many relations, each of which can be connected with the target relation via multiple join paths. Therefore it usually needs to construct many joined relations when searching for good features, which is inefficient in both time and space. This is also verified in our experiments.

There have been some studies on improving efficiency and scalability of ILP algorithms [6, 14, 15]. In [14] an approach was proposed to build multi-relational decision trees from data stored on disks. In [15] the authors proposed an approach that can evaluate packs of queries that share common prefixes simultaneously, in order to improve efficiency by utilizing the shared parts of different queries. This approach shares a similar idea with CrossMine, that is, if the evaluation of different features share common prefixes in join paths, no repeated computation needs to be performed. However, the approach in [15] requires the queries to be known beforehand, which is not true in the searching process of most ILP algorithms. In contrast, CrossMine can propagate tuple IDs freely among different relations at any time, which enables it to decide the direction of search on the fly, in order to avoid aimless search in the huge hypothesis space of relational classification.

In [6] the authors proposed an approach called *label propagation*, which propagates class labels along n-to-1 joins for evaluating predicates. However, for join paths that involve 1-to-n or n-to-n relationships, it cannot find the numbers of positive and negative target tuples satisfying each predicate. For example, some target tuples may join with 10 tuples in relation R, while some others may join with only 1 tuple. If only class labels are propagated, some target tuples may have much higher weight than others, which leads to high skew in evaluating predicates or attributes. Therefore, label propagation cannot be applied to most real world databases which contain 1-to-n or n-to-n relationships. In contrast, tuple ID propagation can be performed on any join paths, thus can help to identify important features in multiple inter-connected relations.

Besides ILP, probabilistic approaches [44, 102, 94] are also popular for multi-relational classification. Probabilistic relational model [44, 102] is an extension of Bayesian networks for handling relational data, which can integrate the advantages of both logical and probabilistic approaches for knowledge representation and reasoning. In [94] an approach is proposed to integrate ILP and statistical modelling for document classification and retrieval. There has also been some study on mining data stored in multiple databases [109], especially on analysis of local patterns and combination of such patterns. CrossMine can be used for identifying local patterns, and can be adapted to multi-database environments.

3.7 Discussions

Till now it is assumed that the dataset can fit in main memory, so that random access can be performed on tuples. In some real applications the dataset cannot fit in main memory, and is stored in a relational database in secondary storage. However, this will not affect the scalability of CrossMine. In this section we show that all the operations of CrossMine can be performed efficiently on data stored on disks.

Tuple ID propagation is the basic operation of CrossMine. When data is in main memory, a set of tuple IDs associated with a relation R are stored in a separate array. When data cannot fit in main memory, we can store a set of tuple IDs as an attribute of R. Since CrossMine limits the fan-out of tuple ID propagation (Section 3.3.3), the number of IDs associated with each tuple is limited, thus the IDs can be stored as a string of fixed or variable length. In CrossMine, only joins between keys or foreign-keys are considered (Section 3.2). An index can be created for every key or foreign key. When propagating IDs from R_1 to R_2, only the tuple IDs and the two joined attributes are needed. If one of them can fit in main memory, this propagation can be done efficiently. Otherwise, a join operation can be performed between R_1 and R_2 to find joinable tuples and propagated IDs.

Suppose tuple IDs have been propagated to a relation R, and the best predicate (or attribute) on R needs to be identified. If all attributes of R are categorical, then the numbers of positive and negative target tuples satisfying every predicate can be calculated by one sequential scan. With this sequential scan, we can also generate simple statistics (sum, average, etc.) for every target tuple and every numerical attribute. The best aggregation literal can be found by these statistics. For a numerical attribute A, suppose a sorted index has been built on A. Then a sorted scan on A is needed to find the best literal on A. If this index and the tuple IDs can fit in main memory, this can be done efficiently. In general, both CrossMine-Rule and CrossMine-Tree can be easily adapted to data stored on disks.

3.8 Summary

Multi-relational classification is an important problem in data mining and machine learning involving large, real databases. It can be widely used in many disciplines, such as financial decision making and medical research. Traditional ILP approaches are usually inefficient and unscalable for databases with complex schemas because they repeated join different relations when evaluating many features in different relations. In this chapter we propose a set of novel tools for multi-relational classification, including tuple ID propagation, an efficient and flexible approach for virtually joining relations, and new definitions for predicates and decision-tree nodes in multi-relational environments. Based on these techniques, we propose two efficient and accurate approaches for multi-relational classification: CrossMine-Rule and CrossMine-Tree. They can perform efficient and effective search in the huge search space, and identify good elements for building accurate classifiers. Experiments show that CrossMine-Rule and CrossMine-Tree are highly scalable comparing with the traditional ILP approaches, and also achieve high accuracy. These features make them appropriate for multi-relational classification in real world databases.

Chapter 4

Multi-relational Clustering with User Guidance

After studying multi-relational classification, we turn to a slightly more complicated task: Multi-relational clustering. The main difference between clustering and classification is that, clustering does not require pre-labelled data. It partitions the data objects into groups, so that objects in each group are more similar to each other than objects in different groups. This is a challenging task in multi-relational environments, because it is both more difficult and more expensive to model the similarity between objects. In this chapter we will describe our method for clustering objects using multi-relational data, which is guided by the user's goal of clustering.

4.1 Overview

In clustering process, objects are grouped according to their similarities between each other, and thus the most essential factor for generating reasonable clusters is a good measure for object similarity. Most existing clustering approaches work on a single table, and the similarity measure is pre-defined. In most traditional clustering approaches [61, 65, 75, 88], the similarity measure is defined as the Euclidean distance or cosine similarity. In some recent studies of projected clustering in high-dimensional space [1, 2], the similarity between two objects is defined based on the top-k dimensions on which they are most similar.

Clustering in multi-relational environments is a different story. A relational database usually contains information of many aspects. For example, a database of computer science department usually contains the following types of information about students: Demographic information, courses, grades, research groups, advisors, publications, etc. If all these relations and attributes are used indiscriminately, it is unlikely that reasonable clusters can be generated.

On the other hand, the user usually has some expectation for clustering, which is of crucial importance in multi-relational clustering. We call such expectation as "the goal of clustering". For example, the user may want to cluster students based on their research interests (as in the database shown in Figure 4.1). It may not be meaningful to cluster them based on their attributes like phone number, ssn, and residence address,

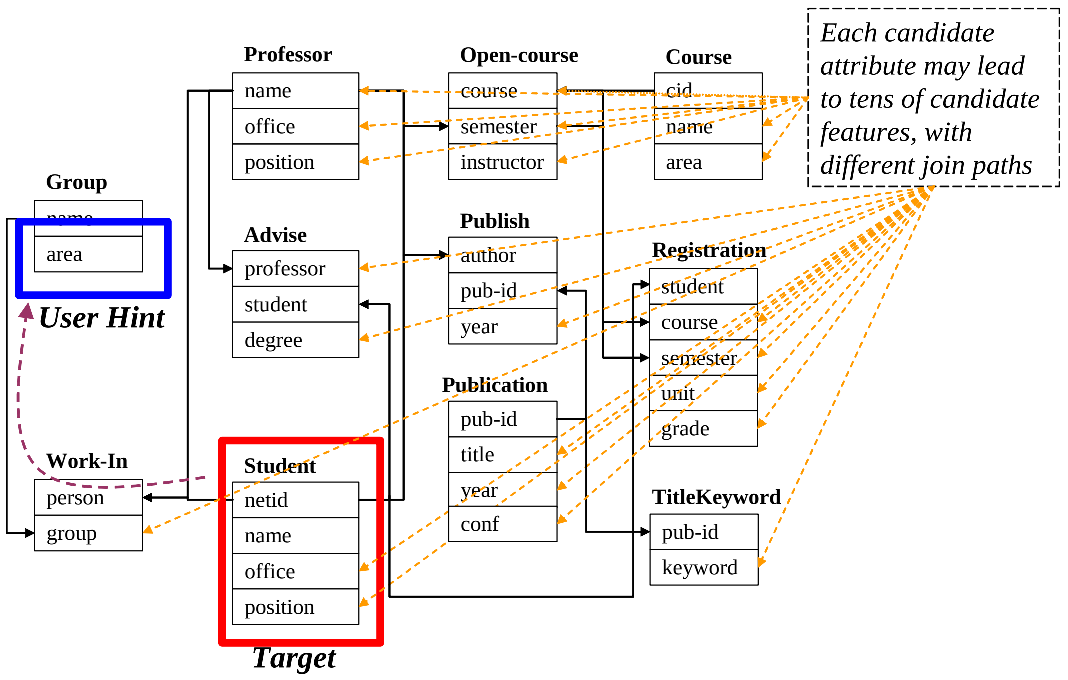


Figure 4.1: Schema of the CS Dept database. This database contains information about the professors, students, research groups, courses, publications, and relationships among them.

even they reside in the same table. The crucial information, such as a student's advisor and publications, are stored in some attributes in a few relations.

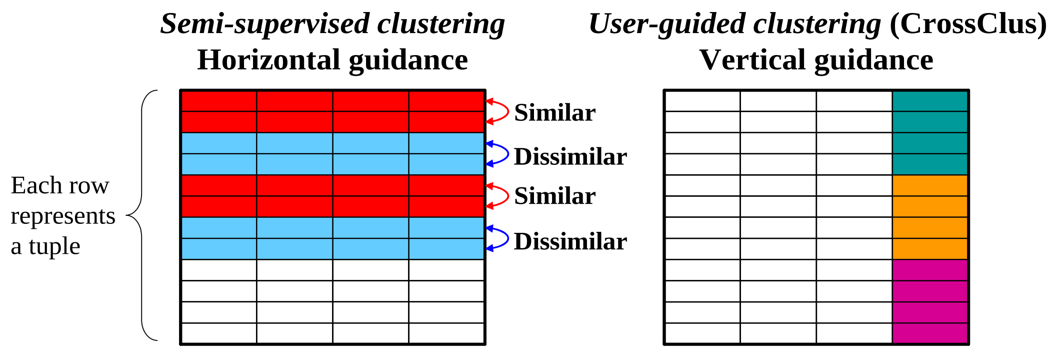
In general, the greatest challenge in multi-relational clustering is how to let the user express her clustering goal, and how to cluster target objects according to this goal. This problem has been studied in clustering in individual tables. We will summarize their work, and then analyze the new challenges in multi-relational environments.

Semi-supervised clustering [10, 69, 103, 110] is proposed to incorporate user-guidance into the clustering process. It takes user-provided information, which is a set of "must-link" or "cannot-link" pairs of objects. It generates clusters according to these constraints, so that most "must-link" pairs are put into the same cluster, and most "cannot-link" pairs are not. However, semi-supervised clustering requires the user to provide a reasonably large set of constraints, each being a pair of similar or dissimilar objects. Since different users have different clustering goals, each user has to provide her own constraints, and this requires good knowledge about the data and the clustering goal. The situation is even worse in relational databases. In order to judge whether two objects in a relational database should be clustered together, a user needs to consider the many tuples joinable with them, instead of only comparing their attributes. Therefore, it is very burdensome for a user to provide a high quality set of constraints, which is critical to the quality of clustering.

In this chapter we introduce a new methodology called *user-guided multi-relational clustering*, and an approach called CrossClus for clustering objects using multi-relational information. User guidance is of crucial importance because it indicates the user's needs. The main goal of user-guided clustering is to utilize user guidance, but minimize user's effort by removing the burden of providing must-link and cannot-link pairs of objects. In fact even a very simple piece of user guidance, such as `clustering students based on research areas`, could provide essential information for effective multi-relational clustering. Therefore, we adopt a new form of user guidance, — *one or a small set of pertinent attributes*, which is very easy for users to provide.

Example 1. In the CS Dept database in Figure 4.1, the goal is to cluster students according to their research areas. A user query for this goal could be "`CLUSTER` $Student$ `WITH` $Group.area$ `AND` $Publication.conference$".

As shown in Figure 4.2, the constraints of semi-supervised clustering contains pairs of similar (must-link) or dissimilar (cannot-link) records, which is "horizontal guidance". In contrast, CrossClus uses "vertical guidance", which are the values of one or more attributes. Both types of guidance provide crucial information for modelling similarities between objects. It is usually very time-consuming to provide the "horizontal guidance", as much information in multiple relations is usually needed to judge whether two objects are similar. In comparison, it is much easier for a user to provide "vertical guidance" by observing the database schema (and possibly some data) and selecting one (or a few) attribute that she thinks to be most pertinent.



Semi-supervised clustering
Horizontal guidance
User-guided clustering (CrossClus)
Vertical guidance
Each row represents a tuple
Similar
Dissimilar
Similar
Dissimilar

Figure 4.2: Constraints of semi-supervised clustering vs. user hint of user-guided clustering

The user hint in user-guided clustering shows the user's preference. Although such hint is represented by one or a small set of attributes, it is fundamentally different from class labels in classification. First, the user hint is used for indicating similarities between objects, instead of specifying the class labels of objects. For example, in Example 1 the user hint "$Publication.conference$" has more than 1000 different values, but a user probably wants to cluster students into tens of clusters. Therefore, CrossClus should not treat the user hint as class labels, or cluster students only with the user hint. Instead, it should learn a similarity model from the user hint and use this model for clustering. Second, the user hint may provide very limited information, and many other factors should be incorporated in clustering. Most users are only capable or willing to provide a very limited amount of hints, and CrossClus needs to find other pertinent attributes for generating reasonable clusters. For example, in Example 1 many other attributes are also highly pertinent in clustering students by research areas, such as advisors, projects, and courses taken.

In user-guided multi-relational clustering, the user hint (one or a small set of attributes) are not sufficient for clustering, because they usually provide very limited information or have inappropriate properties (*e.g.*, too many or too few values). Therefore, the crucial challenge is how to identify other pertinent features for clustering, where a feature is an attribute either in the target relation or in a relation linked to the target relation by a join path. Here we are faced with two major challenges.

Challenge 1: How to measure the pertinence of a feature? There are usually a very large number of features (as in Figure 4.1), and CrossClus needs to select pertinent features among them based on user hints. There have been many measures for feature selection [33, 37, 54, 78] designed for classification or trend analysis. However, the above approaches are not designed for measuring whether different features cluster objects in similar ways, and they can only handle either only categorical features or only numerical ones. We propose a novel method for measuring whether two features cluster objects in similar ways, by comparing the inter-object similarities indicated by each feature. For a feature f, we use the similarity between each pair of objects indicated by f (a vector of $N \times N$ dimensions for N objects) to represent f. When comparing two features f and g, the cosine similarity of the two vectors for f and g is used. Our feature selection measure captures the most essential information for clustering, — inter-object similarities indicated by features. It treats categorical and numerical features uniformly as both types of features can indicate similarities between objects. Moreover, we design an efficient algorithm to compute similarities between features, which never materializes the $N \times N$ dimensional vectors and can compute similarity between features in linear space and almost linear time.

Challenge 2: How to search for pertinent features? Because of the large number of possible features, an exhaustive search is infeasible, and CrossClus uses a heuristic method to search for pertinent features. It starts from the relations specified in user hint, and gradually expands the search scope to other relations, in order to confine the search procedure in promising directions and avoid fruitless search.

After selecting pertinent features, CrossClus uses three methods to cluster objects: (1) Clarans [88], a scalable sampling based approach, (2) k-means [75], a most popular iterative approach, and (3) agglomerative hierarchical clustering [61], an accurate but less scalable approach. Our experiments on both real and synthetic datasets show that CrossClus successfully identifies pertinent features and generates high-quality clusters. It also shows the high efficiency and scalability of CrossClus even for large databases with complex schemas.

The remaining of this chapter is organized as follows. We discuss the related work in Section 4.2 and present the preliminaries in Section 4.3. Section 4.4 describes the approach for feature search, and Section 4.5 presents the approaches for clustering objects. Experimental results are presented in Section 4.6, and this study is concluded in Section 4.7.

4.2 Related Work

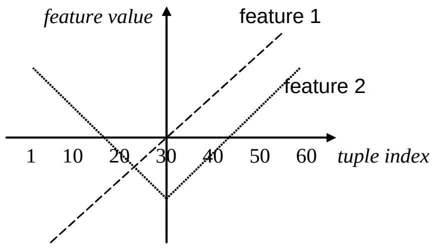
Clustering has been extensively studied for decades in different disciplines including statistics, pattern recognition, database, and data mining, using probability-based approaches [22], distance-based approaches [65, 75], subspace approaches [2, 1], and many other types of approaches.

Clustering in multi-relational environments has been studied in [46, 67, 68], in which the similarity between two objects are defined based on tuples joinable with them via a few joins. However, these approaches are faced with two challenges. First, it is usually very expensive to compute similarities between objects, because an object is often joinable with hundreds or thousands of tuples. Second, a multi-relational feature can be created from an attribute in a relation R and a join path connecting the target relation and R. There are usually a large number of multi-relational features in a database (such as in Figure 4.1), generated from different attributes with different join paths. They cover different aspects of information (*e.g.*, research, grades, address), and only a small portion of them are pertinent to the user's goal. However, all features are used indiscriminately in above approaches, which is unlikely to generate desirable clustering results.

Semi-supervised clustering [10, 69, 103, 110] can perform clustering under user's guidance, which is provided as a set of "must-link" and "cannot-link" pairs of objects. Such guidance is either used to reassign tuples [103] to clusters, or to warp the distance metric to generate better models [10, 69, 110]. However, it is burdensome for each user to provide such training data, and it is often difficult to judge whether two tuples belong to same cluster since all relevant information in many relations needs to be shown to user. In contrast, CrossClus allows the user to express the clustering goal with a simple query containing one or a few pertinent attributes, and will search for more pertinent features across multiple relations.

Selecting the right features is crucial for cross-relational clustering. Although feature selection has been extensively studied for both supervised learning [54], unsupervised learning [37, 78], the existing approaches may not be appropriate for multi-relational clustering. The most widely used criteria for selecting numerical features include Pearson Correlation [55] and least square error [78]. However, such criteria focus on trends of features, instead on how they cluster tuples, because two features with very different trends may cluster tuples in similar ways. For example, the values of 60 tuples on two features are shown in Figure 4.3. According to the above criteria, the two features have correlation of zero. We create 6 clusters according to each feature, as in the right side of Figure 4.3. If two tuples are in the same cluster according to feature 1, they have chance of 50% be in the same cluster according to feature 2; and vice versa. This chance is much higher than when feature 1 and feature 2 are independent (e.g., if feature 2 has random values, then the chance is only 16.7%). Therefore, feature 1 and feature 2 are actually highly correlated, and they cluster tuples in rather similar ways, although they are "uncorrelated" according to the correlation measures mentioned above.

For categorical features, the most popular criteria include information gain [77] or mutual information [54]. However, it is difficult to define information gain or mutual information for multi-relational features, because a tuple has multiple values on a feature. Therefore, we propose a new definition for similarity between features, which focuses on how features cluster tuples and is independent with types of features.



Clusters by feature 1	Clusters by feature 2
{1, …, 10}	{1, …, 5, 56, …, 60}
{11, …, 20}	{6, …, 10, 51, …, 55}
{21, …, 30}	{11, …, 15, 46, …, 50}
{31, …, 40}	{16, …, 20, 41, …, 45}
{41, …, 50}	{21, …, 25, 36, …, 40}
{51, …, 60}	{26, …, 30, 31, …, 35}

Figure 4.3: Two "uncorrelated" features and their clusters

4.3 Problem Definitions

We use R_x to represent a relation in the database. The goal of multi-relational clustering is to cluster objects in a *target relation* R_t, using information in other relations linked with R_t in the database. The tuples of R_t are the objects to be clustered and are called *target tuples*. CrossClus accepts user queries that contain a *target relation, and one or a small set of pertinent attribute(s).* Consider the query "`CLUSTER` *Student* `WITH` *Group.area* `AND` *Publication.conference*" in Example 1, which aims at clustering students according to their research areas. *Student* is the target relation, and *Group.area* and *Publication.conference* are the pertinent attributes. Since the pertinent attributes provide crucial but very limited information for clustering, CrossClus searches for other pertinent features, and groups the target tuples based on all these features. Different relations are linked by joins. As in some other systems on relational databases (such as DISCOVER [60]), only joins between keys and foreign-keys are considered by CrossClus. Other joins are ignored because they do not represent strong semantic relationships between objects. Due to the nature of relational data, CrossClus uses a new definition for objects and features, as described below.

4.3.1 Multi-Relational Features

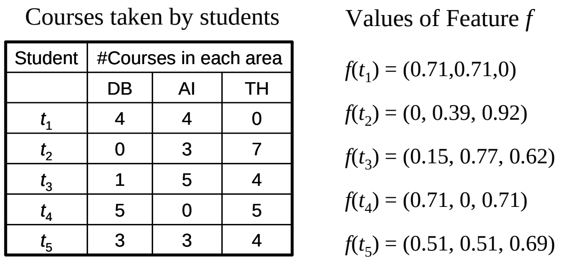
Unlike objects in single tables, an object in relational databases can be joinable with multiple tuples of a certain type, and thus has multiple values on a multi-relational feature. For example, a feature may represent courses taken by students, and a student has multiple values on this feature if he takes more than one course.

In CrossClus a multi-relational feature f is defined by a join path $R_t \bowtie R_1 \bowtie \cdots \bowtie R_k$, an attribute $R_k.A$ of R_k, and possibly an aggregation operator (*e.g.*, average, count, max). f is formally represented by $[f.joinpath, f.attr, f.aggr]$. A feature f is either a *categorical feature* or a *numerical one*, depending on whether $R_k.A$ is categorical or numerical. For a target tuple t, we use $f(t)$ to represent t's value on f. We use $T_f(t)$ to represent the set of tuples in R_k that are joinable with t, via $f.joinpath$.

If f is a categorical feature, $f(t)$ represents the distribution of values among $T_f(t)$.

Definition 4 ***(Values of categorical features)*** *Let f be a categorical feature, and suppose $f.attr$ has l values $v_1, \ldots, v_l$. For a target tuple t, suppose there are $n(t, v_i)$ tuples in $T_f(t)$ that have value v_i ($i = 1, \ldots, l$). t's value on f, $f(t)$, is an l-dimensional vector $(f(t).p_1, \ldots, f(t).p_l)$, where*

$$f(t).p_i = \frac{n(t, v_i)}{\sqrt{\sum_{j=1}^{l} n(t, v_j)^2}} \tag{4.1}$$


Courses taken by students

Student	#Courses in each area		
	DB	AI	TH
t_1	4	4	0
t_2	0	3	7
t_3	1	5	4
t_4	5	0	5
t_5	3	3	4

Values of Feature f

$f(t_1)$ = (0.71,0.71,0)

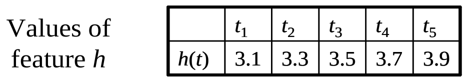
$f(t_2)$ = (0, 0.39, 0.92)

$f(t_3)$ = (0.15, 0.77, 0.62)

$f(t_4)$ = (0.71, 0, 0.71)

$f(t_5)$ = (0.51, 0.51, 0.69)

(a) A categorical feature (areas of courses)


Values of feature h

	t_1	t_2	t_3	t_4	t_5
$h(t)$	3.1	3.3	3.5	3.7	3.9

(b) A numerical feature (average grade)

Figure 4.4: Values of a categorical feature f and a numerical feature h

From Definition 4 it can be seen that $f(t).p_i$ is proportional to the number of tuples in $T_f(t)$ having value v_i, and $f(t)$ is a unit vector. For example, suppose *Student* is the target relation in the CS Dept database.

Consider a feature $f = [Student \bowtie Register \bowtie OpenCourse \bowtie Course, area, \text{null}^1]$ (where $area$ represents the areas of the courses taken by a student). An example is shown in Figure 4.4 (a). A student t_1 takes four courses in database and four in AI, thus $f(t_1) =$ (database:0.5, AI:0.5). In general, $f(t)$ represents t's relationship with each value of $f.attr$. The higher $f(t).p_i$ is, the stronger the relationship between t and v_i is. This vector is usually sparse and only non-zero values are stored.

If f is numerical, then f has a certain aggregation operator (average, count, max, ...), and $f(t)$ is the aggregated value of tuples in the set $T_f(t)$, as shown in Figure 4.4 (b).

4.3.2 Tuple Similarity

Tuple Similarity for Categorical Features

In CrossClus a categorical feature does not simply partition the target tuples into disjoint groups. Instead, a target tuple may have multiple values on a feature. Consider a categorical feature f that has l values $v_1, \ldots, v_l$. A target tuple t's value on f is a unit vector $(f(t).p_1, \ldots, f(t).p_l)$. $f(t).p_i$ represents the proportion of tuples joinable with t (via $f.joinpath$) that have value v_i on $f.attr$.

Because we require the similarity between two values to be between 0 and 1, *we define the similarity between two target tuples t_1 and t_2 w.r.t. f as the cosine similarity between the two unit vectors $f(t_1)$ and $f(t_2)$*, as follows.

Definition 5 *The similarity between two tuples t_1 and t_2 w.r.t. f is defined as*

$$sim_f(t_1, t_2) = \sum_{k=1}^{l} f(t_1).p_k \cdot f(t_2).p_k \tag{4.2}$$

We choose cosine similarity because it is a most popular method for measuring the similarity between vectors. In Definition 5 two values have similarity one if and only if they are identical, and their similarity is always non-negative because they do not have negative values on any dimensions.

Tuple Similarity for Numerical Features

Different numerical features have very different ranges and variances, and we need to normalize them so that they have equal influences in clustering. For a feature h, similarity between two tuples t_1 and t_2 w.r.t. h is defined as follows.

[1] The null value indicates that no aggregation operator is used in this feature.

Definition 6 *Suppose σ_h is the standard deviation of tuple values of numerical feature h. The similarity between two tuples t_1 and t_2 w.r.t. h is defined as*

$$sim_h(t_1, t_2) = 1 - \frac{|h(t_1) - h(t_2)|}{\sigma_h}, \; if \; |h(t_1) - h(t_2)| < \sigma_h; \quad 0, otherwise. \tag{4.3}$$

That is, we first use *Z score* to normalize values of h, so that they have mean of zero and variance of one. Then the similarity between tuples are calculated based on the Z scores. If the difference between two values is greater than σ_h, their similarity is considered to be zero.

4.3.3 Similarity Vectors

When searching for pertinent features during clustering, both categorical and numerical features need to be represented in a coherent way, so that they can be compared and pertinent features can be found. In clustering, the most important information carried by a feature f is how f clusters tuples, which is conveyed by the similarities between tuples indicated by f. Thus in CrossClus a feature is represented by its *similarity vector*, as defined below.

Definition 7 *(**Similarity Vector**) Suppose there are N target tuples $t_1, \ldots, t_N$. The similarity vector of feature f, $\mathbf{v}^f$, is an N^2 dimensional vector, in which ${\mathbf{v}^f}_{iN+j}$ represents the similarity between t_i and t_j indicated by f.*

The similarity vector of feature f in Figure 4.4 (a) is shown in Figure 4.5. In the visualized chart, the two horizontal axes are tuple indices, and the vertical axis is similarity. The similarity vector of a feature represents how it clusters target tuples, and is used in CrossClus for selecting pertinent features according to user guidance. Similarity vector is a universal method for representing features, no matter whether they are categorical or numerical.

4.4 Finding Pertinent Features

4.4.1 Similarity between Features

Given the user guidance, CrossClus selects pertinent features based on their relationships to the feature specified by user. Similarity between two features is measured based on their similarity vectors, which represent the ways they cluster tuples. If two features cluster tuples very differently, their similarity vectors are very different, and their similarity is low. Therefore, when measuring the similarity between two features

$$\begin{pmatrix} 1 & 0.278 & 0.654 & 0.5 & 0.727 \\ 0.278 & 1 & 0.871 & 0.649 & 0.833 \\ 0.654 & 0.871 & 1 & 0.545 & 0.899 \\ 0.5 & 0.649 & 0.545 & 1 & 0.848 \\ 0.727 & 0.833 & 0.899 & 0.848 & 1 \end{pmatrix}$$

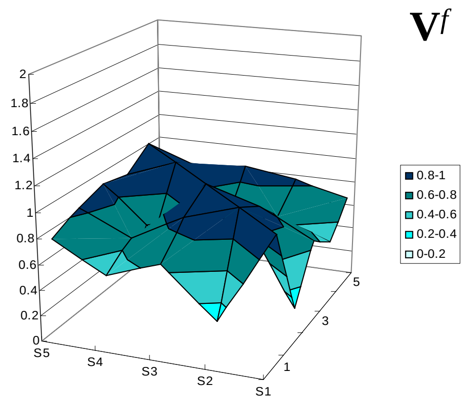


Figure 4.5: Similarity vectors of feature f

f and g, we compare their similarity vectors $\mathbf{v}^f$ and $\mathbf{v}^g$. The similarity between two features is defined as the cosine similarity between their similarity vectors[2]. This definition can be applied on any types of features.

Definition 8 *The similarity between two features f and g is defined as*

$$sim(f,g) = \frac{\mathbf{v}^f \cdot \mathbf{v}^g}{|\mathbf{v}^f| \cdot |\mathbf{v}^g|}, \qquad (\ |\mathbf{v}^f| = \sqrt{\mathbf{v}^f \cdot \mathbf{v}^f}\ ) \tag{4.4}$$

$sim(f,g) = 1$ if and only if $\mathbf{v}^f$ and $\mathbf{v}^g$ differ by a constant ratio, and $sim(f,g)$ is small when most pairs of tuples that are similar according to f are dissimilar according to g. Figure 4.6 shows the values of two features $f = [Student \bowtie Register \bowtie OpenCourse \bowtie Course,\ area,\ \text{null}]$ (areas of courses taken by each student) and $g = [Student \bowtie WorkIn,\ group,\ \text{null}]$ (research group of each student). Their similarity vectors $\mathbf{v}^f$ and $\mathbf{v}^g$ are shown in Figure 4.7. To compute $sim(f,g)$, one needs to compute the inner product of $\mathbf{v}^f$ and $\mathbf{v}^g$, as shown in Figure 4.7.

Feature f			
t ID	DB	AI	TH
1	0.5	0.5	0
2	0	0.3	0.7
3	0.1	0.5	0.4
4	0.5	0	0.5
5	0.3	0.3	0.4

Feature g			
t ID	Info sys	Cog sci	Theory
1	1	0	0
2	0	0	1
3	0	0.5	0.5
4	0.5	0	0.5
5	0.5	0.5	0

Figure 4.6: The values on two features f and g

[2]Most clustering algorithms group tuples that are relatively similar, instead of considering absolute similarity values. Thus we use cosine similarity to ignore magnitude of similarity values.

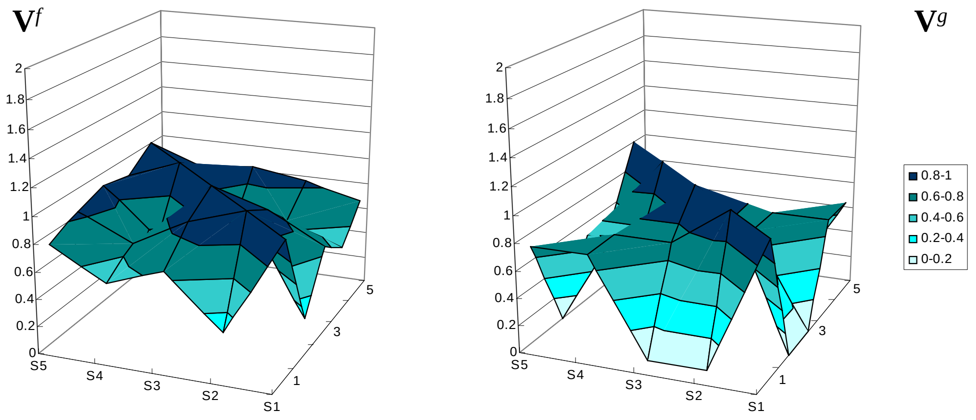


Figure 4.7: Inner product of $\mathbf{v}^f$ and $\mathbf{v}^g$

4.4.2 Efficient Computation of Feature Similarity

It is very expensive to compute the similarity between two features in the brute-force way following Definition 8. Actually we cannot even afford storing N^2 dimensional vectors for many applications. Therefore, we design an efficient algorithm to compute $sim(f, g)$ without materializing $\mathbf{v}^f$ and $\mathbf{v}^g$, in linear or almost linear time.

Similarity between Categorical Features

We first describe the approach for computing similarities between categorical features. The main idea is to convert the hard problem of computing the inner product of two similarity vectors of N^2 dimensions, into an easier problem of computing similarities between feature values, which can be solved in linear time. Similarities between tuples are defined according to their relationships with the feature values, as in Definition 5. Similarities between feature values can be defined similarly according to their relationships with the tuples.

Definition 9 *The similarity between value v_k of feature f and value v_q of feature g is defined as*

$$sim(f.v_k, g.v_q) = \sum_{i=1}^{N} f(t_i).p_k \cdot g(t_i).p_q \tag{4.5}$$

The definitions of tuple similarity and feature value similarity are similar and symmetric to each other. Because of the symmetric definitions, we can convert the inner product of $\mathbf{v}^f$ and $\mathbf{v}^g$ from summation of tuple similarities into that of feature value similarities.

Lemma 3 *Convert summation of tuple similarities into that of feature value similarities.*

$$\mathbf{v}^f \cdot \mathbf{v}^g = \sum_{i=1}^{N}\sum_{j=1}^{N} sim_f(t_i, t_j) \cdot sim_g(t_i, t_j) \;\; = \;\; \sum_{k=1}^{l}\sum_{q=1}^{m} sim(f.v_k, g.v_q)^2 \tag{4.6}$$

PROOF.

$$\begin{aligned}\mathbf{v}^f \cdot \mathbf{v}^g = & \sum_{i=1}^{N}\sum_{j=1}^{N}\left[\sum_{k=1}^{l} f(t_i).p_k \cdot f(t_j).p_k\right]\left[\sum_{q=1}^{m} g(t_i).p_q \cdot g(t_j).p_q\right] \\ = & \sum_{k=1}^{l}\sum_{q=1}^{m}\left[\sum_{i=1}^{N} f(t_i).p_k \cdot g(t_i).p_q\right]\left[\sum_{j=1}^{N} f(t_j).p_k \cdot g(t_j).p_q\right] \\ = & \sum_{k=1}^{l}\sum_{q=1}^{m} sim(f.v_k, g.v_q)^2\end{aligned}$$

With Lemma 3, to compute the similarity between f and g, we only need to compute $sim(f.v_k, g.v_q)$ for each value v_k of f and v_q of g. We can compute them by scanning the tuples just once. For each tuple t, we get $f(t)$ and $g(t)$. For each value v_k so that $f(t).v_k > 0$, and each v_q so that $g(t).v_q > 0$, we update $sim(f.v_k, g.v_q)$ by adding $f(t).v_k \cdot g(t).v_q$. In this way $sim(f.v_k, g.v_q)$ ($1 \le k \le l$, $1 \le q \le m$) can be computed in one scan. This requires an entry in memory for each $sim(f.v_k, g.v_q)$. In reality most categorical features have no more than hundreds of values. If f has too many values, CrossClus will make an indexed scan on f (scanning tuples having each value of f), and it only needs to maintain an entry for each value of g. In general, $\mathbf{v}^f \cdot \mathbf{v}^g$ can be computed in linear time, with very limited extra space.

Similarity between Categorical and Numerical Features

A different approach is needed for computing similarities involving numerical features. Consider the numerical feature h in Figure 4.4 (b). For simplicity we assume that $t_1, \ldots, t_N$ are sorted according to h. It is expensive to directly compute the inner product of two N^2 dimensional vectors $\mathbf{v}^h$ and $\mathbf{v}^f$. Again we try to convert this problem into a problem that can be solved in linear time. From the definition of tuple similarity w.r.t. numerical features (Section 4.3.2), one can see that only tuples whose values differ by at most one standard deviation have non-zero similarity with each other. When we scan tuples ordered by h, only a subset of tuples need to be considered when scanning each tuple. The main idea of our algorithm is to decompose $\mathbf{v}^h \cdot \mathbf{v}^f$ into two parts, so that one part only depends on each tuple, and the other part contains some statistics of the previously scanned tuples, which can be maintained incrementally. In this way $\mathbf{v}^h \cdot \mathbf{v}^f$ can be computed by one scan.

Because of the symmetric definition of tuple similarity, $sim(t_i, t_j) = sim(t_j, t_i)$ for any feature. Let $\eta(i)$ represent the minimum index j so that $h(t_i) - h(t_j) \le \sigma$. As i increases when scanning tuples, $\eta(i)$ either increases or stays the same. Thus we can find all tuples with $h(t_i) - h(t_j) \leq \sigma$ by maintaining two pointers on t_i and $t_{\eta(i)}$. We have

$$\mathbf{v}^h \cdot \mathbf{v}^f = 2 \sum_{i=1}^{N} \sum_{j=\eta(i)}^{i-1} sim_h(t_i, t_j) \cdot sim_f(t_i, t_j), \tag{4.7}$$

which can be efficiently computed with Lemma 4.

Lemma 4 *For a numerical feature h and categorical feature f, $\mathbf{v}^h \cdot \mathbf{v}^f$ can be computed as*

$$V^h \cdot V^f = 2\sum_{i=1}^{N} \sum_{j=\eta(i)}^{i-1} \left[1 - \left(h(t_i) - h(t_j)\right)\right] \left[\sum_{k=1}^{l} f(t_i).p_k \cdot f(t_j).p_k\right]$$

$$= 2\sum_{i=1}^{N}\sum_{k=1}^{l} \boxed{f(t_i).p_k\left(1-h(t_i)\right)} \left(\boxed{\sum_{j=\eta(i)}^{i-1} f(t_j).p_k}\right) + 2\sum_{i=1}^{N}\sum_{k=1}^{l} \boxed{f(t_i).p_k} \left(\boxed{\sum_{j=\eta(i)}^{i-1} h(t_j) \cdot f(t_j).p_k}\right)$$

Only depend on t_i Statistics of t_j with $j<i$

To compute $\mathbf{v}^h \cdot \mathbf{v}^f$, we scan all tuples in the order of h. When scanning each tuple t_i, we get $f(t_i).p_k \cdot (1 - h(t_i))$ and $f(t_i).p_k$ for each value v_k of f. We also dynamically maintain the two statistics $\sum_{j=\eta(i)}^{i-1} f(t_j).p_k$ and $\sum_{j=\eta(i)}^{i-1} h(t_j) \cdot f(t_j).p_k$.[3] In this way we can compute the inner product with Lemma 4, using one scan on the tuples and very limited extra space.

Similarity Between Numerical Features

The similarity between two numerical features h and g is computed in a different way. Suppose tuples are sorted according to their values on h. According to Definition 6, when scanning tuple t^*, only tuples t with $|h(t) - h(t^*)| < \sigma_h$ and $|g(t) - g(t^*)| < \sigma_g$ need to be considered, which is usually a small subset of tuples. Therefore, we compute $\mathbf{v}^h \cdot \mathbf{v}^g$ in the following way.

As shown in Figure 4.8, we scan tuples in the order of h. A search tree is dynamically maintained, which contains the indices of all tuples t with $h(t^*) - h(t) < \sigma_h$. This tree is sorted and indexed by tuple values on g. When scanning a tuple t^*, it is inserted into the tree, and all tuples t with $h(t^*) - h(t) < \sigma_h$, $|g(t) - g(t^*)| < \sigma_g$ can be easily found in the tree by an indexed search. $\mathbf{v}^h \cdot \mathbf{v}^g$ can be updated with the values of these tuples according to Definitions 6 and 8. Because the number of tuples satisfying $h(t^*) - h(t) < \sigma_h$, $|g(t) - g(t^*)| < \sigma_g$ is usually quite small, $\mathbf{v}^h \cdot \mathbf{v}^g$ can be efficiently computed by one scan on the tuples. This procedure requires $O(N \log N)$ time because it needs to maintain a search tree.

[3]This can be done by adding the corresponding part of newly scanned tuple and removing the corresponding part of those tuples going out of the range.

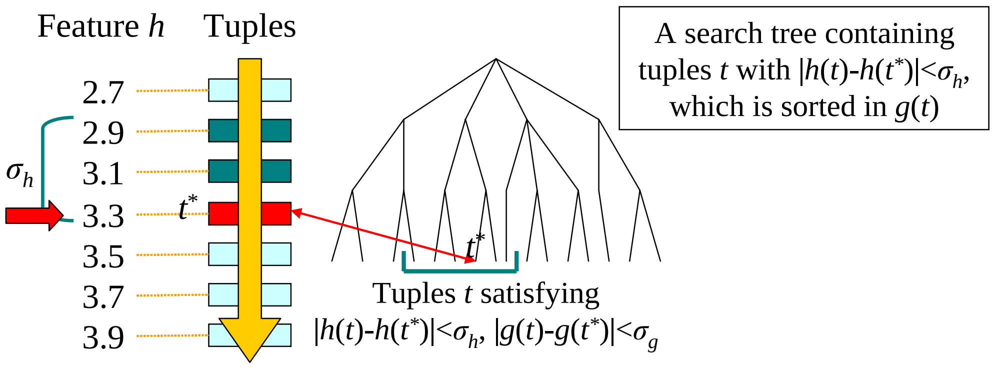

Feature h
Tuples
2.7
2.9
3.1
3.3
3.5
3.7
3.9
σ_h
t^*
A search tree containing tuples t with $|h(t)-h(t^*)|<\sigma_h$, which is sorted in $g(t)$
t^*
Tuples t satisfying $|h(t)-h(t^*)|<\sigma_h$, $|g(t)-g(t^*)|<\sigma_g$

Figure 4.8: Computing similarity between numerical features h and g

Triangle Inequality

Although similarity between two features can usually be computed in linear time, it is still expensive to compute similarities between many features when the number of target tuples is large. We observe that $sim(f, g)$ is actually the cosine of the angle between two vectors $\mathbf{v}^f$ and $\mathbf{v}^g$, which lie in a Euclidean space. Thus we can utilize the *triangle inequality* to further improve efficiency.

According to the law of cosines, the distance between $\mathbf{v}^f$ and $\mathbf{v}^g$ can be computed by

$$|\mathbf{v}^f - \mathbf{v}^g| = \sqrt{|\mathbf{v}^f|^2 + |\mathbf{v}^g|^2 - 2|\mathbf{v}^f||\mathbf{v}^g| sim(f, g)} \tag{4.8}$$

For any three features f, g and h, since $\mathbf{v}^f$, $\mathbf{v}^g$ and $\mathbf{v}^h$ are vectors in Euclidean space, according to the triangle inequality,

$$\begin{aligned} |\mathbf{v}^f - \mathbf{v}^g| &\geq \left| |\mathbf{v}^f - \mathbf{v}^h| - |\mathbf{v}^h - \mathbf{v}^g| \right|, \\ |\mathbf{v}^f - \mathbf{v}^g| &\leq |\mathbf{v}^f - \mathbf{v}^h| + |\mathbf{v}^h - \mathbf{v}^g| \end{aligned} \tag{4.9}$$

With the triangle inequality, before computing the similarity between features f and g, a range can be determined for $sim(f, g)$ using their similarities to other features. This helps save some computation in searching for features similar to a certain feature.

4.4.3 Efficient Generation of Feature Values

When searching for pertinent features, CrossClus needs to generate the values of target tuples on many different features. It is often expensive to join all relations along the join path of a feature. Thus we need an efficient approach for generating feature values. CrossClus uses a technique called *Tuple ID Propagation* [111], which is a way to virtually join tuples in target relation R_t with tuples in another relation R_k, via a certain join path $p = R_t \bowtie R_1 \bowtie \cdots \bowtie R_k$. Its main idea is to propagate the IDs of target tuples from R_t to the relations along the join path one by one, so that for each relation R_i on the join path, each tuple t' of R_i is associated with the IDs of all target tuples joinable with t'. After propagating IDs along a join path, the IDs are stored on each relation on the path and can be further propagated to other relations.

Suppose a feature f is defined by a join path $p = R_t \bowtie R_1 \bowtie \cdots \bowtie R_k$, and an attribute $R_k.A$. To generate the values of each target tuple on f, we first propagate IDs along path p. (If we have already propagated IDs along a prefix of p, those IDs can be used.) The IDs associated with every tuple t' in R_k represent target tuples joinable with t'. From this information we can easily compute the tuples in R_k joinable with each target tuple, then compute the value on f for each target tuple.

4.4.4 Searching for Pertinent Features

Given a clustering task with user guidance, CrossClus searches for pertinent features across multiple relations. There are two major challenges in feature search. (1) *The number of possible features is very large in multi-relational environment.* The target relation R_t can usually join with each relation R via many different join paths, and each attribute in R can be used as a feature. It is impossible to perform exhaustive search in this huge feature space. (2) *Among the many features, some are pertinent to the user query (e.g., a student's advisor is related to her research area), while many others are irrelevant (e.g., a student's demographic information).* It is a challenging task to identify pertinent features while avoiding aimless search in irrelevant regions in the feature space.

To overcome the above challenges, CrossClus tries to confine the search procedure in promising directions. It uses a heuristic approach, which starts from the user-specified features, and then repeatedly searches for useful features in the neighborhood of existing features. In this way it gradually expands the search scope to relevant relations without going deep into random directions.

The large number of possible features cover different aspects of information. For example, some features are about a student's research area, such as her research group or conferences of publications. Some others are about her academic performance, such as her grade or awards. Our experiments show that features in the same aspect usually have high similarities between each other, while features in different aspects usually have low similarities. A user is usually interested in clustering tuples based on a certain aspect of information, and indicates this by a query. Recall the user query in Example 1 where the user wants to cluster students by research areas:

CLUSTER $Student$ WITH $Group.area$ AND $Publication.conference$

To create the initial features for this query, CrossClus searches for the shortest join path from the target relation $Student$ to relation $Group$ and relation $Publication$, and creates a feature f using each path. If f is not qualified (*e.g.*, many target tuples have empty values on f), CrossClus searches for the next shortest join path, and so on. As in Figure 4.1, two initial pertinent features are created: [$Student \bowtie WorkIn \bowtie Group$, $name$, null] and [$Student \bowtie Publish \bowtie Publication$, $conference$, null].

Based on the initial features, CrossClus searches in multiple relations for pertinent features, which have high similarity with the initial features. Because exhaustive search is infeasible, CrossClus uses a heuristic method that has been widely used in ILP methods [12, 96, 111]. This method considers the relational schema as a graph, with relations being nodes and joins being edges. It starts from the node of the initial feature, and explores the surrounding nodes of pertinent features. The best feature in the current search scope is added as a pertinent feature, and the search scope is expanded to all neighbor nodes of those nodes containing pertinent features. In this way it avoids aimless search in the huge feature space by confining the search in promising directions.

CrossClus also uses the *look-one-ahead* strategy in ILP approaches. That is, when it expands the search scope to a relation of relationship R_r (*e.g.*, $Advise$, $WorkIn$, $Publish$ and $Registration$ in Figure 4.1), it automatically expands the scope to the relation of entity referenced by R_r. For example, when it reaches $Advise$ relation from $Student$, it will also search $Professor$ relation. We use this strategy because usually no pertinent features can be found in relations of relationship, although they may serve as bridges to other important features.

Each feature is given a weight between 0 and 1, which is determined by its relationship to the user guidance. The weight of the initial pertinent feature is 1. For each pertinent feature f that is found later, f's weight is determined by its similarity with the initial pertinent features. CrossClus uses the average similarity between f and each initial pertinent feature as the weight of f. Finally all features whose weight are above a threshold $weight_{min}$ are used for clustering.

Figure 4.9 shows the algorithm of feature search. At first the initial features are the only pertinent features. At each step, CrossClus gets all candidate features and evaluates each of them. The *triangle inequality* can be used when possible. For example, suppose there are two initial pertinent features f_1^* and f_2^*, which have high similarity between each other. If a candidate feature f_c has low similarity with f_1^*, then

Algorithm 3 Feature Searching

Input: A relational database D, a set of initial features F.

Output: A set of pertinent features.

```
candidate feature set $C \leftarrow \emptyset$
for each $f \in F$
    add features with join path $f.joinpath$ to $C$
    for each relation $R$ that can be appended to $f.joinpath$
        $p \leftarrow f.joinpath + R$
        add features with join path $p$ to $C$
    end for
end for
remove features in $F$ from $C$
for each $f \in C$
    compute the similarity between $f$ and each initial feature
end for
repeat
    $f \leftarrow$ feature with maximum weight in $C$
    if $f.weight \geq weight_{min}$ then add $f$ to $F$
    remove $f$ from $C$
    for each relation $R$ that can be appended to $f.joinpath$
        $p \leftarrow f.joinpath + R$
        add features with join path $p$ to $C$
    end for
    for each $f \in C$
        compute the similarity between $f$ and each initial feature
    end for
until $(1 - \varepsilon)$ target tuples are covered by at least $H$ features
return $F$
```

Figure 4.9: Algorithm **Feature Searching**

we may be able to infer that f_c cannot have sufficient similarity with f_2^* and should not be considered. In each step the candidate feature with the highest weight, f_c^*, is added to the set of pertinent features, and new candidate features related to f_c^* will be evaluated in the next step. We say a tuple is *covered* by a feature if it has non-empty value on this feature. The algorithm stops when most target tuples have been covered by at least H features.

4.5 Clustering Tuples

Given a group of features $f_1, \ldots, f_L$, CrossClus groups the target tuples into clusters that contain tuples similar to each other. We will first introduce the similarity measure between tuples, then introduce the three clustering algorithms: Clarans, k-means, and agglomerative clustering.

4.5.1 Similarity Measure

The similarity between target tuples w.r.t. each feature has been defined in Section 4.3.2 and 4.3.2, and such similarity has been used for selecting features for clustering. The same definition is used for measuring tuple similarity in clustering. Consider two target tuples t_1 and t_2, which have values on L features $f_1, \ldots, f_L$. Each feature f_i has weight $f_i.weight$.

Definition 10 *The similarity between t_1 and t_2 is defined as the weighted average of their similarity on each feature.*

$$sim(t_1, t_2) = \frac{\sum_{i=1}^{L} sim_{f_i}(t_1, t_2) \cdot f_i.weight}{\sum_{i=1}^{L} f_i.weight} \tag{4.10}$$

4.5.2 Clustering Method 1: CLARANS

Clarans [88] is a sampling based k-medoids clustering [65] approach, and is a popular clustering algorithm for objects in non-Euclidean spaces because it only requires similarity measure between tuples. The main idea of Clarans is to consider the whole space of all possible clusterings as a graph, and use randomized search to find good clustering in this graph. A clustering (a set of clusters covering all target tuples) is evaluated based on the average similarity between each example to its closest medoid. Clarans starts with k initial medoids and constructs k clusters. In each round an existing medoid is replaced by a new medoid that is randomly picked. If the replacement leads to better clustering, the new medoid is kept; otherwise the clusters remain unchanged. This procedure is repeated until the clusters remain unchanged for a certain number of rounds (25 rounds in our experiments). Clarans is efficient and scalable in most cases as the clusters usually become stable after tens of iterations.

4.5.3 Clustering Method 2: K-Means Clustering

In k-medoids clustering the similarity between a target tuple t and a cluster C is defined as the similarity between t and the medoid of C, which is another target tuple. In k-means clustering [75] the similarity between t and C is the average similarity between t and all tuples in C. The similarity measure in k-means clustering is more accurate, because it is often difficult to find a representative medoid in k-medoids clustering. However, in k-means clustering it is usually unaffordable to compute the similarity between many pairs of individual tuples, and we need to be able to calculate the "average point" of a cluster of target tuples.

Suppose we want to compute the average similarity between a tuple t and a cluster C. There are L features used: $f_1, \ldots, f_L$, each being categorical or numerical. According to Definition 10, the similarity between two tuples is a weighted average of their similarity on each feature. Thus we discuss how to compute the average similarity between t and C based on each feature.

For a categorical feature f with l values, the average similarity between t and C is

$$sim_f(t, C) = \frac{\sum_{t' \in C} \sum_{k=1}^{l} f(t).p_k \cdot f(t').p_k}{|C|} = \sum_{k=1}^{l} f(t).p_k \cdot \frac{\sum_{t' \in C} f(t').p_k}{|C|} \tag{4.11}$$

Thus we can compute the "average point" of C as a tuple $t_{\bar{C}}$, such that $f(t_{\bar{C}}).p_k = \frac{\sum_{t' \in C} f(t').p_k}{|C|}$. The similarity between t and $t_{\bar{C}}$ on f is just the average similarity between t and C on f.

If f is a numerical feature, the average similarity cannot be computed easily because the similarity measure (Section 4.3.2) involves absolute values. However, as each tuple has a simple real value on a numerical feature, the numerical features just form a Euclidean space, and we can just take the arithmetic average as in regular k-means clustering, *i.e.*, $f(t_{\bar{C}}) = \frac{\sum_{t' \in C} f(t')}{|C|}$.

With the definition of the "average point", we can apply k-means algorithm, – repeatedly assigning each tuple to its closest cluster and recomputing the "average point" of each cluster. k-means clustering is often more time-consuming than Clarans for the following reason. The categorical features in CrossClus are set-valued, and each target tuple usually has only a small number of values on each set-valued feature. However, the "average point" of a cluster often has a large number of values, which are the union of the values of each tuple. This makes it expensive to compute the similarity between each target tuple and the "average point" of each cluster.

4.5.4 Clustering Method 3: Agglomerative Hierarchical Clustering

Both Clarans and k-means divide all target tuples into clusters and repeatedly refine the clusters. We use agglomerative hierarchical clustering [61], which generates clusters by merging similar tuples. Initially it uses each tuple as a cluster, and computes the similarity between each pair of tuples. Then it keeps merging the most similar pairs of clusters, until no clusters are similar enough or a certain number of clusters remain. When measuring the similarity between two clusters, we use the similarity between the "average point" of the two clusters as in Section 4.5.3.

After clustering target tuples, CrossClus provides the results to the user, together with information about each feature. From the join paths and attributes of the features, the user can know the meaning of clustering, which helps her understand the clustering results.

4.6 Experimental Results

We report comprehensive experiments on both synthetic and real databases. All tests were done on a 2.4GHz Pentium-4 PC with 1GB memory, running Windows XP. The following parameters are used in CrossClus: $H = 10$, $weight_{min} = 0.4$, and $\varepsilon = 0.05$ in Algorithm 3. All approaches are implemented using Microsoft Visual Studio.Net.

We compare CrossClus with three other approaches: Baseline, Proclus, and RDBC. (1) The Baseline algorithm clusters data by the initial feature only, using Clarans. (2) Proclus [1] is the state-of-the-art subspace clustering approach that works on single tables. To apply Proclus, we first convert relational data into single tables. For each clustering query, we perform a breadth-first search of depth three from the initial features in the schema graph, and use every attribute in every relation as a feature. This generates a table with 50 to 100 features for each query. (3) RDBC [67, 68] is a recent clustering approach for first-order representations. It uses the similarity measure in [39], which measures the similarity of two objects x and y by the similarity between the objects joinable with x and those with y, which are further defined by objects joinable with them. This is very expensive because each object may be joinable with many objects. RDBC can utilize different clustering algorithms, and we choose Clarans as in CrossClus. When measuring similarities between two objects, only directly related objects (objects of depth 0) are used, because it becomes prohibitively slow with larger depth.

Proclus and RDBC are slightly modified to utilize the user guidance. In Proclus the initial features are forced to be included in the feature set of every cluster. In RDBC, the objects related to a target tuple t via the join paths of initial features are added to the set of related objects of t.

4.6.1 Measure of Clustering Accuracy

Validating clustering results is very important for evaluating an approach. In most of our datasets the target tuples are manually labelled according to their semantic information. For example, we labelled the research areas of each professor in CS Dept dataset using their web sites. Jaccard coefficient [101] is a popular measure for evaluating clustering accuracy, which is the number of pairs of objects in same cluster and with same label, over that of pairs of objects either in same cluster or with same label. Because an object in our datasets may have multiple labels, but can only appear in one cluster, there may be many more pairs of objects with same label than those in same cluster. Therefore we use a variant of Jaccard coefficient. We say two objects are correctly clustered if they share at least one common label. The accuracy of clustering is defined as the number of object pairs that are correctly clustered over that of object pairs in same cluster. Higher accuracy tends to be achieved when number of clusters is larger. Thus we let each approach generate the same number of clusters.

4.6.2 Clustering Effectiveness

In this experiment we test whether CrossClus can produce reasonable clusters under user guidance. Three real datasets are used: The CS Dept dataset, the DBLP dataset, and the Movie dataset.

CS Dept dataset

CS Dept database[4] (as shown in Figure 4.1) is collected from web sources of Dept. of CS, UIUC. It has 10 relations and 4505 tuples. The clustering goal is to cluster professors according to their research areas. There are 45 professors, and we label their research areas according to the department web site. Fourteen areas are considered: Theory, artificial intelligence, operating systems, databases, architecture, programming languages, graphics, networking, security, human-computer interaction, software engineering, information retrieval, bioinformatics, and computer-aided design. On average each professor belongs to 1.51 areas.

We test three user guidances: (1) clustering professors based on research groups, (2) based on courses they have taught, and (3) based on both (1) and (2). In all three tests CrossClus selects the following features: research group, courses taught, conferences of publications, keywords in publication titles. When both research groups and courses are used as guidance, the last two features are assigned higher weights, because they are similar to both features used as guidance.

Six approaches are compared: CrossClus with k-means, Clarans, and agglomerative clustering, Baseline, PROCLUS, and RDBC. We let each approach group the professors into 14 clusters, as there are 14 research areas. All approaches finish within 5 seconds. The clustering accuracies are shown in Figure 4.10. One can see that CrossClus achieves much higher accuracy than the other approaches. This is mainly because CrossClus selects a good set of pertinent features and assigns reasonable weights to features. In comparison, PROCLUS and RDBC use all possible features indiscriminately and cannot generate clusters that meet the user's needs, and the Baseline approach can only use a small set of features. Among the three clustering methods used by CrossClus, agglomerative clustering achieves highest accuracy, mainly because it performs more computation, — it repeatedly computes similarity between every pair of clusters and always merges the most similar clusters.

Some clusters found by CrossClus-Agglomerative are shown below:

(*Theory*): J. Erickson, S. Har-Peled, L. Pitt, E. Ramos, D. Roth, M. Viswanathan.

(*Graphics*): J. Hart, M. Garland, Y. Yu.

[4]http://dm1.cs.uiuc.edu/csuiuc_dataset/. All relations contain real data, except the *Register* relation which is randomly generated.

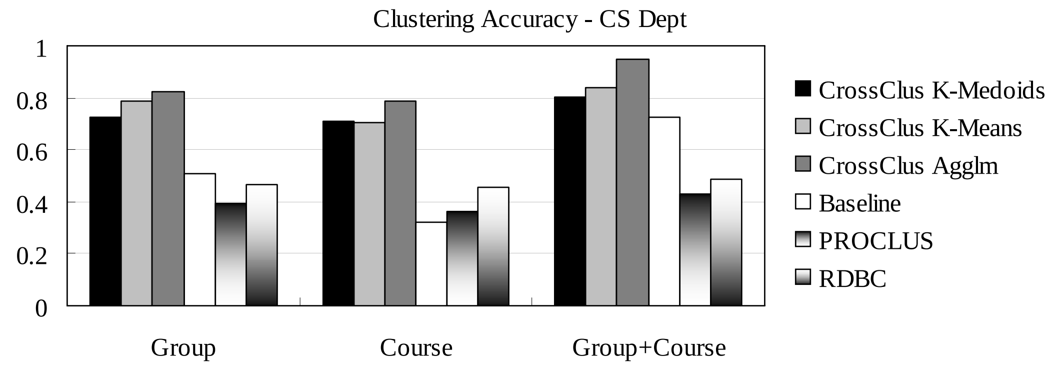


Figure 4.10: Clustering Accuracy on CS Dept dataset

(*Database*): K. Chang, A. Doan, J. Han, M. Winslett, C. Zhai.

(*Numerical computing*): M. Heath, T. Kerkhoven, E. de Sturler.

(*Networking & QoS*): R. Kravets, M. Caccamo, J. Hou, L. Sha.

(*Artificial Intelligence*): G. Dejong, M. Harandi, J. Ponce, L. Rendell.

(*Architecture*): D. Padua, J. Torrellas, C. Zilles, S. Adve, M. Snir, D. Reed, V. Adve.

(*Operating Systems*): D. Mickunas, R. Campbell, Y. Zhou.

DBLP dataset

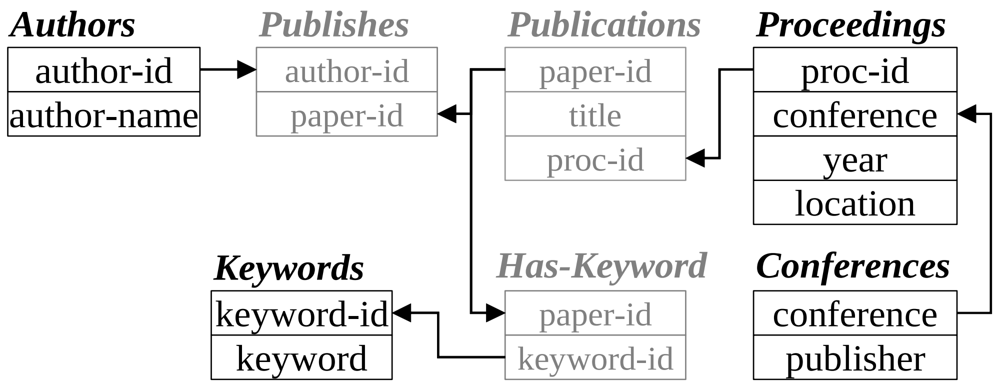


Figure 4.11: The schema of the DBLP dataset. This database contains information about authors, publications (and keywords in their titles), and conferences.

The DBLP dataset is extracted from the XML data of DBLP [32], whose schema is shown in Figure 4.11. We focus our analysis on productive authors, and the goal is to cluster authors according to their research areas. We only keep conferences that have been held for at least 8 times and authors with at least 12 papers.

There are 4170 authors, 2517 proceedings, 154 conferences, and 2518 keywords in the dataset. We analyze the research areas of 400 most productive authors. For each of them, we find her home page and infer her research areas from her research interests (or her publications if no research interests specified). On average each author is interested in 2.15 areas. We let each approach group the 400 authors into 14 clusters, as there are 14 research areas. The other authors are not clustered but may serve as feature values.

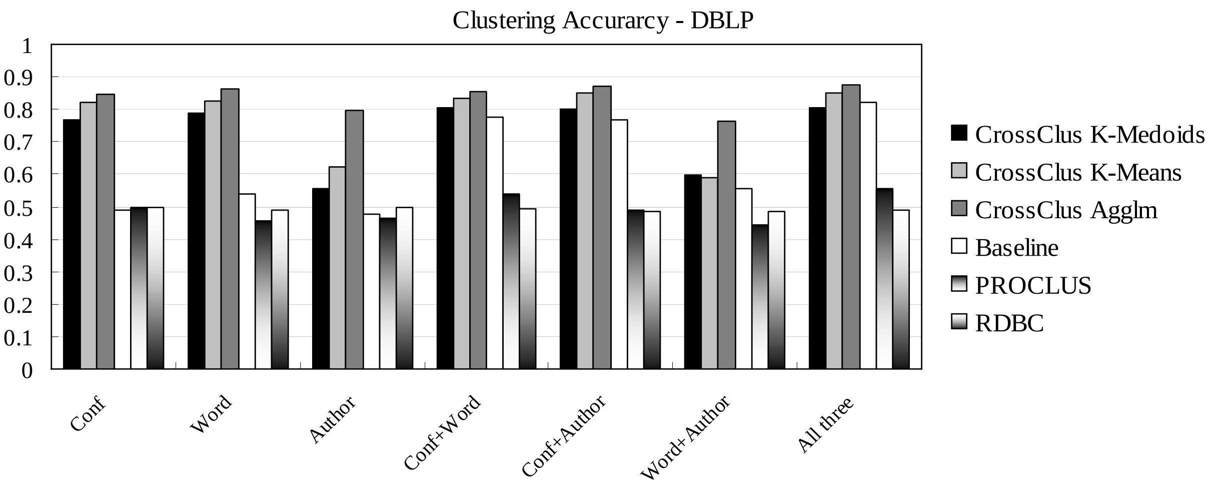


Figure 4.12: Clustering Accuracy on DBLP dataset

Three initial features are used as guidance: (1) Conferences of papers, (2) keywords in paper titles, and (3) coauthors. We test the all combinations of one, two and three initial features. Four features are selected by CrossClus, including the above three and proceedings of publications. However, not all four features are used when there is not enough information in user guidance. For example, when only keywords in paper titles is used as guidance, CrossClus only selects proceedings of publications besides the feature in user guidance.

The results of the six approaches are shown in Figure 4.12. CrossClus achieves highest accuracy, and agglomerative clustering is still most accurate. Again it usually achieves higher accuracy when using more user guidance. Baseline algorithm also achieves high accuracy when three initial features are used, because there is no other important features besides the three initial features. However, it is very difficult for a user to specify all pertinent features, and our goal is to minimize user effort while still generating reasonable clusters.

The running time of each approach is shown in Table 4.1. One can see that CrossClus with Clarans is most efficient, and CrossClus with *k*-means is not very efficient because the "average point" of a cluster may have many values on each categorical feature (*e.g.*, keywords or coauthors), which makes it expensive to compute its similarity with other tuples.

	CrossClus			Baseline	Proclus	RDBC
	Clarans	k-means	Agglm			
1 initial feature	54.6	151.5	105.6	7.51	78.51	342.9
2 initial features	61.8	181.2	124.7	13.8	72.28	415.9
3 initial features	73.3	202.1	129.7	21.4	72.83	542.8

Table 4.1: Running time (seconds) on DBLP dataset

Movie dataset

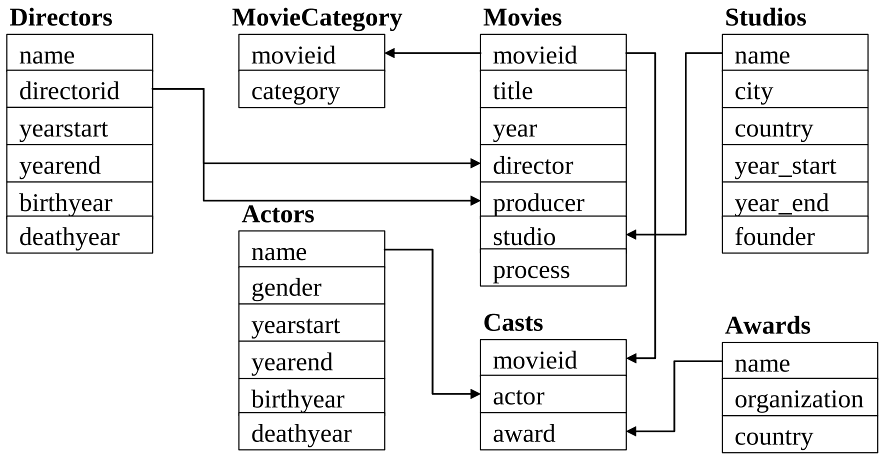


Figure 4.13: Schema of Movie dataset

We test CrossClus on Movie dataset from UCI KDD Archive[5], which has 7 relations and 60817 tuples, containing 11284 movies, 6773 actors and 3112 directors. Since we do not have labels on this dataset due to lack of expertise, for each clustering goal we select the *standard feature set*, which is *the set of features containing information directly related to the task*. Using this standard feature set, we create the *standard clustering* for the target tuples using Clarans. The accuracy of a clustering is measured by its similarity with the standard clustering. However, because there exist random factors in clustering procedure, and each tuple is only assigned to one cluster, slight difference in features may lead to large difference in clustering results. Thus we use a more "lenient" measure for clustering accuracy as in [112].

The similarity between two clusterings C and C' is measured by how much the clusters in C and C' overlap with each other. Suppose C has n clusters $c_1, \ldots, c_n$, and C' has n' clusters $c'_1, \ldots, c'_{n'}$. The degree of C subsuming C' is defined as,

$$deg(C \subset C') = \frac{\sum_{i=1}^{n} \max_{1 \le j \le n'}(|c_i \cap c'_j|)}{\sum_{i=1}^{n} |c_i|} \tag{4.12}$$

[5]http://kdd.ics.uci.edu/

$deg(C \subset C')$ is the average proportion of each cluster in C being contained in some cluster in C'. The similarity between C and C' is the average degree that each of them subsumes the other.

$$sim(C, C') = \frac{deg(C \subset C') + deg(C' \subset C)}{2} \tag{4.13}$$

$sim(C, C') = 1$ if and only if C and C' are identical.

Task 1: *Clustering directors according to the genres and styles of their movies.* The query is "`CLUSTER` *Directors* `WITH` *Movies.category*". The movies' styles are related to their categories (genres), countries, studios, and award information. The standard feature set is

1. [$Director \bowtie Movies \bowtie MovieCategory$,*category*, null],
2. [$Director \bowtie Movies$, *studio*, null],
3. [$Director \bowtie Movies \bowtie Studios$, *country*,null],
4. [$Director \bowtie Movies \bowtie Casts$, *award*, null],

The feature set selected by CrossClus includes all above features, plus *Movies.year* and *Movies.process* (b&w, color, silent, etc). The results are shown in Table 4.2. Agglomerative clustering is not tested because it cannot finish in reasonable time on the 3112 directors.

	CrossClus Clarans	CrossClus k-means	Baseline	Proclus	RDBC
Accuracy	0.587	0.590	0.557	0.387	0.192
Runtime (sec)	24.5	39.3	5.53	123	341

Table 4.2: Experiment 1 on Movie dataset

Some directors of famous movies are clustered together by CrossClus with Clarans:

(*Sci.& Fic.*): George Lucas (*Star Wars*), Montgomery Tully (*Battle Beneath the Earth, The Terrornauts*), Robert Wiemer (*Star Trek*), Robert Zemeckis (*Back to the Future 1,2*), Andy Wachowski (*Matrix1,2,3*), ...

(*Romance*): Mike Newell (*Four Weddings and a Funeral, Mona Lisa Smile*), Nora Ephron (*Sleepless in Seattle, You've Got Mail*), Alexander Hall (*Forever Darling, Because You are Mine*), Anthony Minghella (*The English Patient, Mr.Wonderful*), ...

(*Kid and Ani.*): Hamilton Luske (*Cinderella, Peter Pan*), Charles Nichols (*Tom and Jerry*), Roger Allers (*The Lion King, Aladdin*), John Lasseter (*Finding Nemo, Monster, Inc.*), ...

(*Action*): James Cameron (*Terminator, Alien, True Lies*), Brett Ratner (*Rush Hour 1,2*), Quentin Tarantino (*Kill Bill 1,2*), Tony Scott (*Top Gun, Enemy of the State, Crimson Tide*), ...

Task 2: *Clustering actors according to their eras.* The query is "`CLUSTER` *Actors* `WITH` *Movies.movieyear*". The standard feature set is

1. [$Actors \bowtie Casts \bowtie Movies$, *movieyear*,average],
2. [*Actors*, *yearstart*, average],
3. [*Actors*, *yearend*, average],
4. [*Actors*, *birthyear*, average],
5. [*Actors*, *deathyear*, average],

The feature set selected by CrossClus contains feature 1,2,5, and another feature [$Actors \bowtie Casts \bowtie Movies \bowtie Directors$, *yearend*, average]. The results are shown in Table 4.3.[6] Again we do not test agglomerative clustering on the 6773 actors.

	CrossClus Clarans	CrossClus k-means	Baseline	Proclus	RDBC
Accuracy	0.517	0.570	0.321	0.413	0.380
Runtime (sec)	61.6	25.4	7.16	1008	1080

Table 4.3: Experiment 2 on Movie dataset

From the above experiments it can be seen that CrossClus successfully finds features pertinent to the clustering tasks, and generates more accurate clustering results than the other approaches. This also validates that user guidance plays an important role in clustering.

4.6.3 Scalability Tests

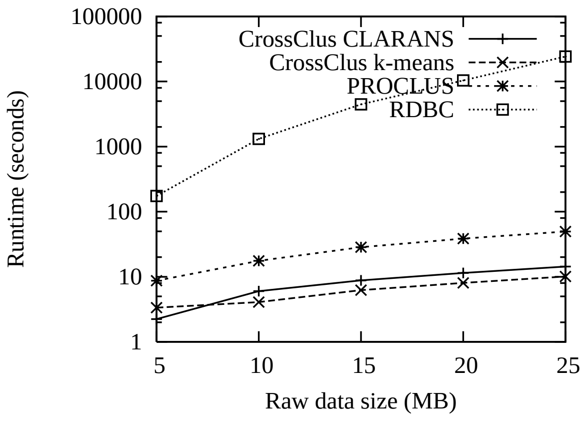
In this experiment we test the scalability of CrossClus (with Clarans and k-means), Proclus, and RDBC. CrossClus with agglomerative clustering is not used because it requires quadratic time and space. We first test the scalability w.r.t. the sizes of databases .We use TPC-H databases[7] of raw data sizes from 5MB to 25MB (number of tuples from 43.5K to 217K). The following query is used "`CLUSTER` *Customer* `WITH` *Orders.totalprice*". For each database, the CrossClus algorithm selects the following four features for clustering: average total price of orders, priority of orders, mktsegments of customers, and regions of customers. Please notice that CrossClus selects some categorical features although the initial feature is numerical.

The running time of CrossClus, Proclus, and RDBC are shown in Figure 4.14 (a) (in log scale), and that of CrossClus is shown in Figure 4.14 (b). It can be seen that CrossClus and Proclus are linearly scalable w.r.t. database size, and CrossClus is substantially more efficient. RDBC becomes unaffordable for large datasets.

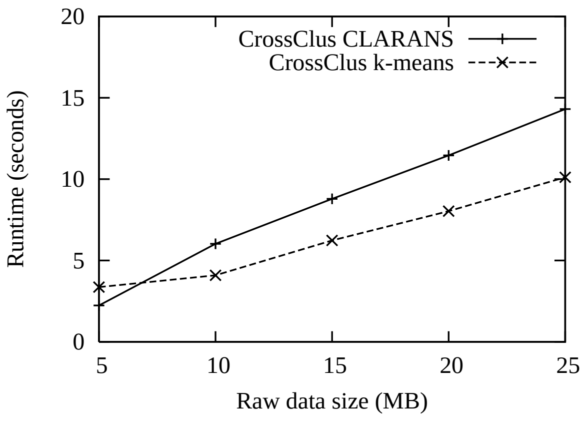
We also test scalability w.r.t. number of relations in databases. We use the data generator used in [111], which randomly generates relational schemas and fills data according to some rules. The expected number

[6]Clustering with a set of numerical features often has low accuracy, because numerical features do not provide clear ways for partitioning tuples. For example, if actors are clustered with their birthyear by two algorithms, it is quite possible that one generates clusters like (1920—1929, 1930—1939, . . .), while the other generates (1925—1934, 1935—1944, . . .).

[7]TPC-H benchmark database. http://www.tpc.org/tpch.



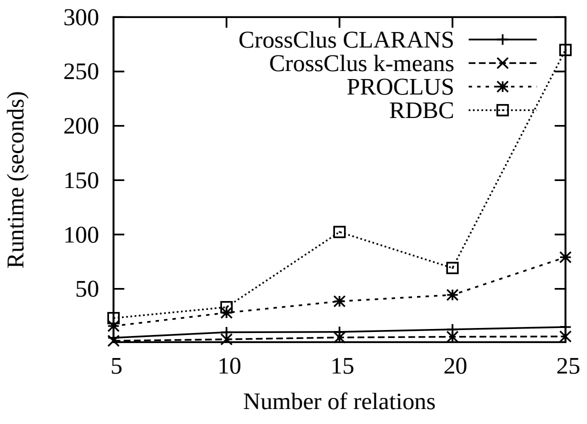
a) All approaches



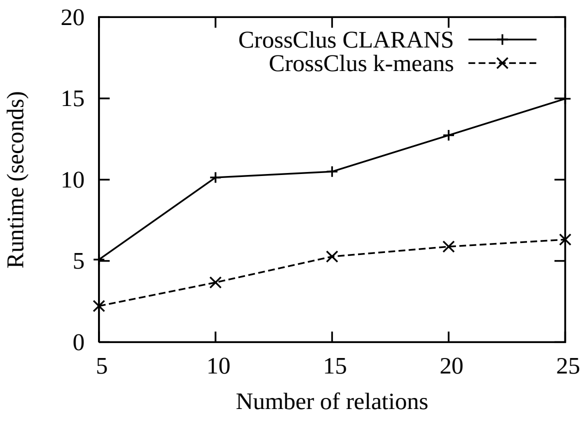
b) CrossClus

Figure 4.14: Scalability on TPC-H

of tuples in each relation is 1000. The results are shown in Figure 4.15. It can be seen that CrossClus is scalable w.r.t. the number of relations.



a) All approaches



b) CrossClus

Figure 4.15: Scalability vs. #relation

4.7 Summary

In this chapter we propose CrossClus, an efficient and effective approach for cross-relational clustering. Because there exist numerous features in a relational database with various semantic meanings, CrossClus employs a new concept: *user-guided clustering*, which selects features and performs clustering based on user's guidance. We propose a new form of user guidance, — one or a small set of pertinent features, which is very easy for user to provide. We also propose a new similarity measure for feature selection based on how they cluster tuples. Our experiments show that CrossClus generates meaningful clusters that match the users' expectation, and achieves high efficiency and scalability. We believe CrossClus represents a promising direction for user guided clustering of data in multiple relations.

Chapter 5

Multi-relational Duplicate Detection

In this chapter we discuss a real-world application of multi-relational data mining: Duplicate detection in relational databases. Duplicate detection aims at finding objects that appear to be different but are actually the same. For example, “Alon Y. Levy” and “Alon Y. HaLevy” may be considered as two different persons, but are actually different names of the same person. The most important challenge of duplicate detection is how to measure the similarity between different objects. This is especially true in relational environments, because two objects can be related in many different ways in a relational database.

In this chapter we study the properties of duplication in relational databases, especially the linkages between duplicate objects. Based on this analysis, we design an approach for duplicate detection based on linkages between different objects.

5.1 Overview

Duplication exists in most databases and data warehouses in the world, which is generated by various reasons such as spelling errors, use of abbreviations, and inconsistent conventions. In many applications substantial amounts of time and money are spent on detecting and removing duplicate tuples in databases. There have been extensive studies on duplicate detection (also known as deduplication, record linkage, merge/purge and object matching) [4, 8, 11, 21, 25, 34, 36, 42, 59, 63, 85, 99, 97, 107]. They employ many different methodologies, including probabilistic approaches [42, 85, 107], string similarity based approaches [4, 21, 25, 59, 63], and machine learning based approaches [11, 34, 36, 99, 97].

Most existing approaches work on a single table containing a set of tuples. To determine whether two tuples are duplicates, their values on each attribute are compared. Different models are used to compute the similarity on each attribute (equivalence/inequivalence, edit-distance, TF/IDF, etc.), and different methods are used to combine similarities on different attributes (joint probability, weighted sum, SVM kernels, etc.). However, most real-world structured data is stored in relational databases. When detecting duplicates in a relational database, although the tuples to be matched are usually stored in one relation, in most cases no single relation can provide sufficient information for detecting duplicates. For example, in the database of a CS department (as in Figure 5.1), in order to detect duplicate tuples in *Student* relation, one will need information in many other relations including their advisors, research groups, courses, and publications.

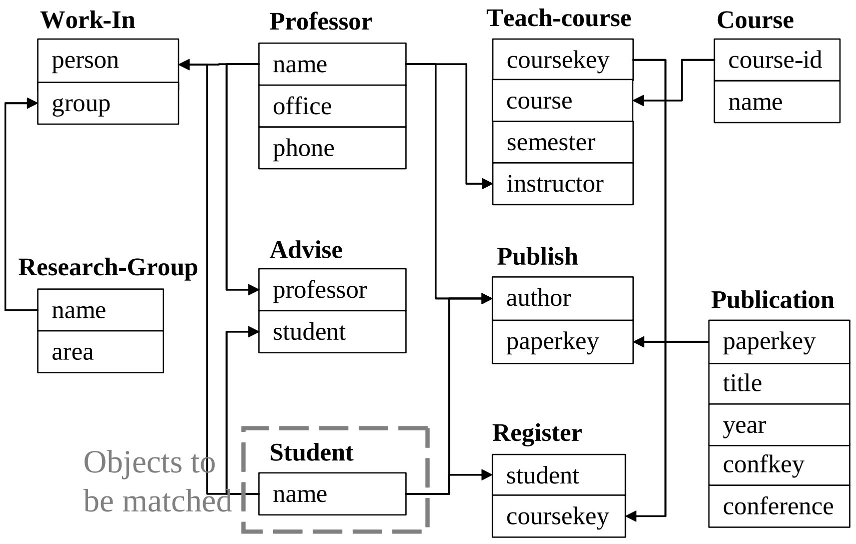


Figure 5.1: Schema of CS department database

Some approaches have been proposed to utilize certain types of relational information. In [4] and [8] approaches are proposed to detect duplicates with concept hierarchies or relational data. When matching two tuples in a certain relation, they consider not only their attributes, but also their "neighbor tuples" (tuples directly joinable with them). In Semex system for personal information management [36], the user can select certain context information such as co-authors and email contacts, in order to detect duplicates among a certain type of tuples (e.g., authors).

Although the above approaches can utilize some relational information, the types of relational information being used are limited to either directly joinable tuples, or those specified by users. On the other hand, a relational database contains very rich linkage information, which cannot be used by above approaches. For example, in the database of a CS department, two students can be linked in many ways, such as they share advisors, take same courses, or one student coauthors with another student's advisor. Some linkages may be very useful in duplicate detection. For example, a student may appear with different names in administration data and publication data (e.g., "Xiaoxin Yin" and "X. Yin"), leading to duplicate tuples in *Student* relation. It is very likely that there exist some linkages between them, such as "X. Yin" coauthors papers with the advisor of "Xiaoxin Yin", as shown in Figure 5.2.

Such duplicates are very common in databases containing data integrated from different sources, and

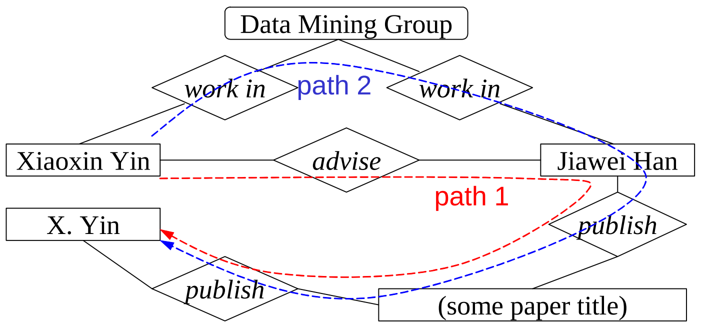


Figure 5.2: Duplicates in CS Dept database

are also introduced by spelling errors or inconsistent conventions, especially in relations input by different people. These duplicates cannot be detected by existing approaches, which only utilize simple relational information such as directly joinable tuples. On the other hand, if we collapse tuples joinable (directly or indirectly) with each student into a set, the useful information may be overwhelmed by the vast amount of useless information, such as the hundreds of classmates of each student. Therefore, it is desirable to design a new approach that considers the linkages between different tuples for duplicate detection.

We study this problem from a new perspective. Instead of studying the "common values" or "common neighbors" of different tuples for duplicate detection, we focus on the linkages between different tuples. We define a path p between two tuples x and y as a list of tuples that starts from x and ends at y, and each two consecutive tuples on p are joinable. For example, in Figure 5.2 "Xiaoxin Yin" and "X. Yin" are connected by two paths, one going through advisor and publication, and the other going through research group and publication.

Paths can capture any kind of linkages in a relational database, and our experiments show that paths provide very useful information in duplicate detection. In Section 5.3 we will present experiments showing that, under certain definition of connectivity based on paths, most tuples have higher connectivity to themselves than to other tuples in the same relation. For example, on average each author in DBLP database [32] has higher connectivity to herself than to 99.9% of authors. Therefore, if an author appears with two different names in all of her papers, the two tuples corresponding to the two names should be highly connected to each other. This property can be used to make accurate judgement on whether two tuples are duplicates.

Based on the above observation, we introduce Relom (*Relational Object Matcher*), a new approach for matching duplicate tuples in relational databases based on paths between tuples. We test two models for measuring strength of connection between tuples, and finally select a random walk model, which measures the probability of walking between two tuples in the graph of tuples. Because different join paths have very different semantic meanings and properties, a uniform random walk strategy may bury the useful information in the piles of useless information. Therefore, we choose a path-based random walk model, in which the probabilities of walking along different *join paths* are independent from each other. We compute the probabilities of walking between different tuples along each join path (there may be many paths between two tuples along a certain join path), and uses machine learning algorithms to combine probabilities along different join paths. Because it is very expensive to compute random walk probabilities between many pairs of tuples in a large database, we propose an approach that can efficiently compute the probability of walking from one tuple to another by computing the probabilities on only half of the join paths.

We perform comprehensive experiments on two real databases. The experiments show that Relom successfully detects hundreds of duplicate authors in DBLP database, with very high precision. Thus we believe Relom is suitable for detecting duplicates in very large relational databases.

The remaining of this chapter is organized as follows. The problem definitions are presented in Section 5.2. We describe Relom's algorithm for duplicate detection in Section 5.4. Empirical evaluation is presented in Section 5.5. We discuss related work in Section 5.6 and conclude this study in Section 5.7.

5.2 Problem Definitions

Consider a relational database D containing a set of relations, and some duplicate tuples in relation R_t. The goal of a multi-relational duplicate detection approach is to detect and merge duplicate tuples in R_t. R_t is called the *target relation*, whose tuples are *target tuples*, which are subjects of duplicate detection. There is usually limited and insufficient information in R_t for detecting duplicates, and one needs to rely on information in other relations. For example, in order to detect duplicate tuples in *Student* relation in Figure 5.1, one needs information about their advisors, publications, courses, research groups, etc.

Relom builds a model for duplicate detection based on a training set, which contains pairs of duplicate target tuples (positive examples) and pairs of distinct target tuples (negative examples). Such a training set is automatically generated as described in Section 5.4.4, instead of manually created.

Relom relies on *paths* between different tuples to measure their connectivity with each other. Here a path p between two tuples t_0 and t_l is a list of tuples $t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_{l-1} \rightarrow t_l$, so that t_i is joinable with t_{i+1} ($0 \leq i \leq l-1$). A path p follows a join path ρ. For example, path 1 in Figure 5.2 follows join path *Student* $\rightarrow$ *Advise* $\rightarrow$ *Professor* $\rightarrow$ *Publish* $\rightarrow$ *Publication* $\rightarrow$ *Publish* $\rightarrow$ *Student*. As in some existing systems on relational databases (e.g., [60]), only joins between keys and their corresponding foreign-keys are considered.

Unlike most existing duplicate detection approaches that measure the similarity between two tuples based on their values on different attributes, Relom uses connectivity between different tuples to detect duplicates, which is defined based on the paths between tuples. This method can utilize linkages between two tuples via arbitrary join paths, thus better captures the inter-connected information in a relational database.

5.3 Closed Loop Property

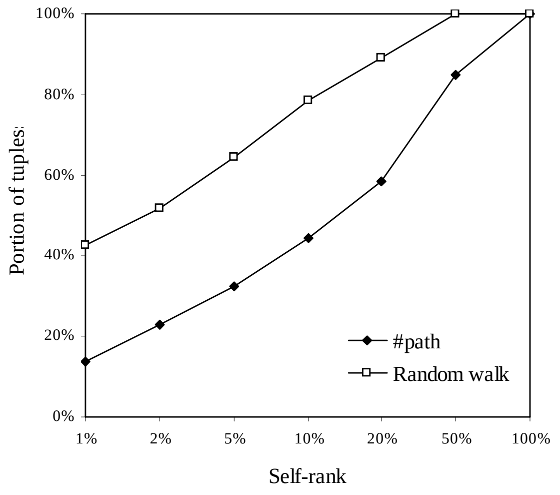
An interesting and useful property is that, most tuples in a relational database are have higher connectivity to themselves than to most other tuples. The underlying reason is that, if a tuple $\hat{t}$ has a path to a tuple t in another relation, then $\hat{t}$ is likely to have more paths to t, which is also shown in social network research [74, 86]. For example, if a student is advised by a professor, she is likely to co-author with the professor, work in the same group with him, and take his classes. Another example is that, if someone publishes a paper in a conference, she is likely to publish more papers in that conference. This leads to the property that a tuple usually has more paths to itself than to other tuples in the same relation.

We call this property as *closed loop property*, which is very useful in duplicate detection. If two tuples are duplicates and should be merged into one tuple, there are usually many paths between them. We verify this property on two datasets: (1) CS Dept dataset [26], which is collected from the web resources of Dept. of Computer Science, UIUC, and (2) DBLP dataset, which is from the XML file of DBLP [32]. The details of these two datasets are provided in Section 5.5.

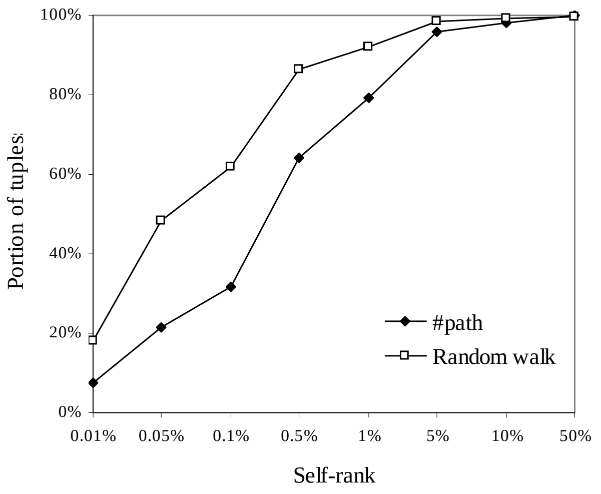
In each dataset, we use two measures to evaluate the connectivity between two tuples $\hat{t}$ and $\hat{t}'$. The first measure is the number of paths between $\hat{t}$ and $\hat{t}'$. The second measure is to perform a random walk from $\hat{t}$ (with equal weight on every join path), and compute the total probability of reaching $\hat{t}'$. This method will be explained in details in Section 5.4. To avoid very expensive computation, we consider paths of length less than 8 (using at most 8 joins). In each measure we do not consider paths like $\hat{t} \to t_1 \to \cdots \to t_1 \to \hat{t}$, because in duplicate detection two duplicate tuples will not be directly joinable with the same tuple.

In CS Dept dataset, for each student $\hat{t}$, we rank all students according to their connectivity with $\hat{t}$ using each measure, from highest to lowest. Then we find the rank of $\hat{t}$ itself in this list (called *self-rank*). The result is shown in Figure 5.3 (a). The x-axis is the self-rank, and the y-axis is the proportion of tuples having such ranks. For example, the left most point of the curve of random walk probability indicates that, about 42% of tuples rank themselves at 1% at most, which means that they have higher connectivity with themselves than at least 99% of tuples. From this figure one can see that random walk probability is a better measure, with which most students rank themselves in top 10%.

For DBLP dataset, because it is very expensive to compute connectivity between a huge number of authors, we randomly select 31,608 authors (10%). Among them we randomly select 500 authors with at



a) CS Dept Dataset



b) DBLP Dataset

Figure 5.3: Closed loop property in CS Dept

least 5 publications. For each selected author $\hat{t}$, we rank the 31,608 authors using their connectivity with $\hat{t}$. The result is shown in Figure 5.3 (b). Again random walk achieves better performance, with which most authors rank themselves in top 0.5%.

The closed loop property is very useful in deduplication. For example, suppose there are two authors "Xiaoxin Yin" and "X. Yin" in DBLP. If they are duplicates, then each of them should rank the other one in top few percent. If they are distinct and unrelated, then the expected rank is 50%. Thus we can perform accurate deduplication if using this property properly.

5.4 Duplicate Detection Algorithm

In general, Relom measures the connectivity between two tuples by combining the random walk probabilities between them along different join paths. In this section we will present (1) our model of path-based random walk, (2) how to efficiently compute probabilities based on such models, and (3) how to build such models from training examples.

The overall procedure of duplicate detection algorithm is as follows:

1. Find $\mathcal{P}$, the set of all join paths that both start and end at the target relation R_t.
2. For each training example that is a pair of tuples $\hat{t_1}$ and $\hat{t_2}$, compute random walk probability from $\hat{t_1}$ to $\hat{t_2}$ and that from $\hat{t_2}$ to $\hat{t_1}$ along each join path in $\mathcal{P}$.

3. For each training example, convert the above probabilities into a vector, whose entries are probabilities along different join paths. Use support vector machines to train a classifier from the vectors of all training examples.

4. For each pair of tuples to be reconciled, compute random walk probabilities between them along different join paths, and use the above classifier to judge whether they are duplicates.

5.4.1 Path-Based Random Walk Model

Consider the graph constructed by using tuples as nodes and joins between tuples as edges. The path-based random walk process starts from a target tuple $\hat{t}$, walks in this graph for a number of steps, and computes the probabilities of walking to other tuples in the database. It is an extension of random walk [79, 92], except that the walk process follows a certain join path, and we are interested in computing probabilities of walking between different tuples, instead of a stable probability distribution on all nodes.

In a relational database different join paths have very different properties and importance. For example, join path *Student* $\rightarrow$ *Work-In* $\rightarrow$ *Research-Group* may play an important role in detecting duplicate students because a student is tightly related to her advisor, but join path *Student* $\rightarrow$ *Register* $\rightarrow$ *Course* may be much less important because one cannot distinguish students by their courses. Therefore, we use a *path-based random walk* model, in which the walk process along each join path is independent from those of other join paths, and the probabilities along each join path are computed separately. Each join path that connects target tuples (i.e., starting and ending at target relation) will be assigned a certain weight by our training process (described in Section 5.4.4). The random walk probabilities along different join paths will be combined in duplicate detection.

The process of random walk is defined as follows. Suppose the process reaches a tuple t in relation R with probability $p(t)$ at a certain step. Suppose R is joinable with relations $R_1, \ldots, R_k$ in the database schema. In our model the total probability of walking from t into all tuples in R_i ($1 \leq i \leq k$) is 1. Suppose t is joinable with m tuples in R_i, then the probability of walking into each of them is $\frac{1}{m}$.

An example is shown in Figure 5.4, in which the random walk process has probability 0.3 of reaching tuple "Jiawei Han" at a certain step. Let $R(t)$ represent the relation in which t belongs to. Let $T_{R \rightarrow R'}(t)$ represent all tuples in R' that are joinable with t which is in R. The probability of walking from t to t' is

$$P[t \rightarrow t'] = \frac{1}{|T_{R \rightarrow R'}(t)|} \tag{5.1}$$

Suppose a path $p = t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_l$. The probability of walking along p is $P[p] = \prod_{i=0}^{l-1} P[t_i \rightarrow t_{i+1}]$.

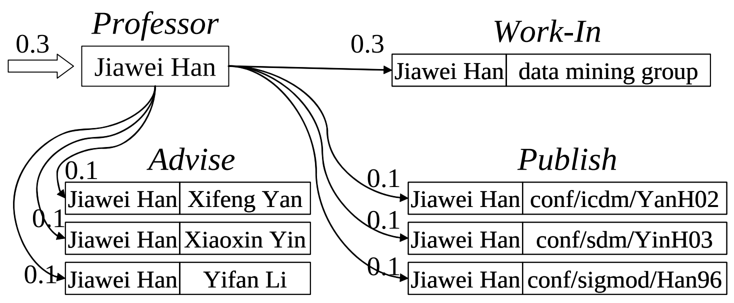


Figure 5.4: Model of path-based random walk

We use $\rho(p)$ to represent the join path that p follows, i.e., $\rho(p) = R(t_0) \to \cdots \to R(t_l)$. We use ρ to represent a join path, and $Q(\rho)$ to represent all paths following ρ. Since we only focus on the connectivity between different target tuples (e.g., authors in DBLP), in the random walk process from a target tuple $\hat{t}$, we are only concerned about the probability of walking from $\hat{t}$ to any target tuple $\hat{t'}$. Because it is very expensive to compute probabilities of walking along very long paths in a very large database (e.g., DBLP), and very long join paths usually indicate weak semantic linkages between tuples, we only compute the probability of random walk within L steps (we use $L = 8$ in our experiments).

Definition 11 ***Random walk probability along a certain join path.*** *Suppose $\hat{t}$ and $\hat{t'}$ are two tuples, and $Q^{\rho}_{\hat{t}\sim\hat{t'}}$ is the set of all paths from $\hat{t}$ to $\hat{t'}$, following join path ρ. We define the* random walk probability from $\hat{t}$ to $\hat{t'}$ along ρ *as $P_\rho[\hat{t} \sim \hat{t'}] = \sum_{p\in Q^{\rho}_{\hat{t}\sim\hat{t'}}} P[p]$.*

The random walk probability between two tuples is defined as a weighted sum of random walk probabilities between them along different join paths. Although weighted sum is a very simple method, it has been used in majority of studies on record linkage [4, 8, 21, 34, 42, 63, 99, 97]. Each join path ρ is assigned a certain weight $w(\rho)$ by the training process, which will be described in Section 5.4.4.

Definition 12 ***Random walk probability between target tuples.*** *Suppose $\hat{t}$ and $\hat{t'}$ are two target tuples, and $\mathcal{P}$ is the set of all join paths that both start and end at the target relation with length no greater than L. We define the* random walk probability from $\hat{t}$ to $\hat{t'}$ *as $P[\hat{t} \sim \hat{t'}] = \sum_{\rho\in\mathcal{P}} w(\rho) \cdot P_\rho[p]$.*

5.4.2 Computing Random Walk Probabilities

It is still very expensive to find all paths between two target tuples t and t' of length no greater than L, and compute their probabilities. In fact a path of length l can be decomposed into two subpaths of length no greater than $\lceil \frac{l}{2} \rceil$. It is much less expensive to compute the probability of all paths of length no greater than $\lceil \frac{L}{2} \rceil$ from either $\hat{t}$ or $\hat{t}'$, and then compute $P[\hat{t} \sim \hat{t}']$ based on such paths. A path can be decomposed as defined below.

Definition 13 ***Balanced decomposition of paths.*** *For a path $p = t_0 \to \cdots \to t_l$, its* balanced decomposition *is two subpaths $p^{\prec}$ and $p^{\succ}$, so that $p^{\prec} = t_0 \to \cdots \to t_{\lceil \frac{l}{2} \rceil}$ and $p^{\succ} = t_{\lceil \frac{l}{2} \rceil} \to \cdots \to t_l$.*

The balanced decomposition of a join path ρ is defined similarly, i.e., if $\rho = R_0 \to \cdots \to R_l$, then $\rho^{\prec} = R_0 \to \cdots \to R_{\lceil \frac{l}{2} \rceil}$ and $\rho^{\succ} = R_{\lceil \frac{l}{2} \rceil} \to \cdots \to R_l$. It can be easily proved that each path (or join path) has one and only one balanced decomposition.

It is easy to show that the probability of walking along path p can be calculated by $P[p] = P[p^{\prec}] \cdot P[p^{\succ}]$. Let $\overleftarrow{p}$ represent the reverse path of p. If we have computed $P[p]$ and $P[\overleftarrow{p}]$ for every path p that starts from $\hat{t}$ or $\hat{t}'$ and has length no greater than $\lceil \frac{L}{2} \rceil$, then we can calculate $P[\hat{t} \sim \hat{t}']$.

For two paths p_1 and p_2, we say p_1 is connected with p_2 if the last tuple of p_1 is same as the first tuple of p_2. We use $p_1 + p_2$ to represent the path of appending p_2 to the end of p_1. Similarly we say join path ρ_1 is connected with ρ_2 if the last relation of ρ_1 is same as the first relation of ρ_2, and use $\rho_1 + \rho_2$ to represent the join path of appending ρ_2 to the end of ρ_1. We use $p_1 \bowtie p_2$ to represent p_1 is connected with p_2, and $0 \leq |p_1| - |p_2| \leq 1$. $\rho_1 \bowtie \rho_2$ is defined similarly.

Theorem 1 *Suppose $\mathcal{P}^l$ is the set of join paths starting at the target relation with length no greater than l. Suppose Q_t^{ρ} is the set of all paths p starting at t following join path ρ. Suppose $P[p]$ and $P[\overleftarrow{p}]$ are known for every path p in Q_t^{ρ} and $Q_{t'}^{\rho}$, for every $\rho \in \mathcal{P}^{\lceil \frac{L}{2} \rceil}$,*

$$P[\hat{t} \sim \hat{t}'] = \sum_{\rho_1 \in \mathcal{P}^{\lceil \frac{L}{2} \rceil}, \rho_2 \in \mathcal{P}^{\lfloor \frac{L}{2} \rfloor}, \rho_1 \bowtie \overleftarrow{\rho_2}} w(\rho_1 + \overleftarrow{\rho_2}) \times \left(\sum_{p_1 \in Q_{\hat{t}}^{\rho_1}, p_2 \in Q_{\hat{t}'}^{\rho_2}, p_1 \bowtie \overleftarrow{p_2}} P[p_1] \cdot P[\overleftarrow{p_2}] \right) \tag{5.2}$$

Proof. *Each path (or join path) has exactly one balanced decomposition. From Definition 13, it can be easily proved that if $p_1 \bowtie p_2$ (or $\rho_1 \bowtie \rho_2$), then p_1 and p_2 (or ρ_1 and ρ_2) are a balanced decomposition of $p_1 + p_2$ (or $\rho_1 + \rho_2$). Thus for a path p of length at most L that starts at t and ends at t', $p^{\prec} \in \bigcup_{\rho \in \mathcal{P}^{\lceil \frac{L}{2} \rceil}} Q_t^{\rho}$, and $\overleftarrow{p^{\succ}} \in \bigcup_{\rho \in \mathcal{P}^{\lfloor \frac{L}{2} \rfloor}} Q_{t'}^{\rho}$. And for any $p_1 \in \bigcup_{\rho \in \mathcal{P}^{\lceil \frac{L}{2} \rceil}} Q_t^{\rho}, p_2 \in \bigcup_{\rho \in \mathcal{P}^{\lfloor \frac{L}{2} \rfloor}} Q_{t'}^{\rho}$, if $p_1 \bowtie \overleftarrow{p_2}$, then $p_1 + \overleftarrow{p_2}$ is a path from t to t' of length at most L. From the above one-to-one mapping, it can be shown that*

$$\begin{aligned}
P[t \sim t'] &= \sum_{\rho \in \mathcal{P}} w(\rho) \cdot \sum_{p \in Q^L_{\hat{t} \to \hat{t'}}} P[p] \\
&= \sum_{\rho \in \mathcal{P}} w(\rho) \cdot \sum_{p \in Q^L_{\hat{t} \to \hat{t'}}} P[p^{\prec}] \cdot P[p^{\succ}] \\
&= \sum_{\rho_1 \in \mathcal{P}^{\lceil \frac{L}{2} \rceil}, \rho_2 \in \mathcal{P}^{\lfloor \frac{L}{2} \rfloor}, \rho_1 \bowtie \overleftarrow{\rho_2}} w(\rho_1 + \overleftarrow{\rho_2}) \times \\
&\quad \left(\sum_{p_1 \in Q^{\rho_1}_{\hat{t}}, p_2 \in Q^{\rho_2}_{\hat{t'}}, p_1 \bowtie \overleftarrow{p_2}} P[p_1] \cdot P[\overleftarrow{p_2}] \right)
\end{aligned}$$

■

Theorem 1 shows that, in order to compute the random walk probability within L steps between a set of target tuples $\hat{t}_1, \ldots, \hat{t}_n$ (which are the target tuples appearing in the training set), we only need to compute the following information: for any path p whose length is no greater than $\lceil \frac{L}{2} \rceil$ and starts from $\hat{t}_i$ $(1 \leq i \leq n)$, compute $P[p]$ and $P[\overleftarrow{p}]$.

This can be done by a depth-first search starting from target relation R_t in the graph of database schema. For each relation R and each qualified join path from R_t to R, we maintain entries like: (1) $P_\rho[\hat{t_i} \sim t]$, the probability of going from target tuple $\hat{t_i}$ to tuple t in R via join path ρ; and (2) $P_{\overleftarrow{\rho}}[t \sim \hat{t_i}]$, the probability of going from t to $\hat{t_i}$ via $\overleftarrow{\rho}$. Only non-zero entries are kept, which are stored in a hashtable and can be accessed or modified in constant time. When the depth-first search goes from relation R to R', the following process is used to compute the random walk probabilities on R'. Suppose ρ_1 is the join path currently used for R, and ρ_2 is the join path of appending R' to ρ_1. For each tuple t in R that have non-zero random walk probability along ρ_1, and each tuple t' in R_2 that is joinable with t, we can calculate $P_{\rho_2}[\hat{t_i} \sim t']$ and $P_{\overleftarrow{\rho_2}}[t' \sim \hat{t_i}]$, as shown in Algorithm 1.

Here we analyze the improvement on efficiency by using the above algorithm to compute random walk probabilities. Suppose Relom needs to compute random walk probabilities between N pairs of target tuples, which are candidates of duplicates detected by some cheap measures [21, 52, 63, 76] (as described in Section 5.4.4). Suppose each tuple in the database is joinable with c tuples on average $(c > 1)$. If we compute random walk probabilities on all paths of lengths at most L, the computational cost is $O(Nc^L)$. If we compute random walk probabilities on paths of lengths at most $\lceil \frac{L}{2} \rceil$, the cost is only $O(Nc^{\lceil \frac{L}{2} \rceil})$, and the cost of computing random walk probabilities between all pairs of candidates is also $O(Nc^{\lceil \frac{L}{2} \rceil})$. Thus the algorithm proposed above can improve efficiency with a magnitude of $c^{\lfloor \frac{L}{2} \rfloor}$ compared with the naive approach.

Algorithm 4 Path-based random walk

Input: A relational database D with a target relation R_t, and a set of target tuples T that appear in the training set.

Output: For any $\hat{t} \in T$, the random walk probability for any path that starts from $\hat{t}$ or ends at $\hat{t}$ and has length no greater than $\lceil \frac{L}{2} \rceil$.

```
Procedure
initialize all random walk probabilities in all relations to 0
for each $\hat{t_i} \in R_t$
    $P_\phi[\hat{t_i} \sim \hat{t_i}] \leftarrow 1$; //where $\phi$ is the empty join path
end for
stack $S \leftarrow$ empty
push $(R_t, \phi)$ into $S$
while($S$ not empty)
    $(R, \rho_1) \leftarrow$ pop($S$)
    for each relation $R'$ joinable with $R$
        $\rho_2 \leftarrow$ append $R'$ to the end of $\rho_1$
        for each $t$ in $R$ and $\hat{t_i}$ in $R_t$ that $P_{\rho_1}[\hat{t_i} \sim t] > 0$
            for each $t'$ in $R'$ joinable with $t$
                $P_{\rho_2}[\hat{t_i} \sim t'] \leftarrow P_{\rho_2}[\hat{t_i} \sim t'] + P[t \to t'] \cdot P_{\rho_1}[\hat{t_i} \sim t]$
            end for
        end for
        for each $t$ in $R$ and $\hat{t_i}$ in $R_t$ that $P_{\overleftarrow{\rho_1}}[t \sim \hat{t_i}] > 0$
            for each $t'$ in $R'$ joinable with $t$
                $P_{\overleftarrow{\rho_2}}[t' \sim \hat{t_i}] \leftarrow P_{\overleftarrow{\rho_2}}[t' \sim \hat{t_i}] + P[t' \to t] \cdot P_{\overleftarrow{\rho_1}}[t \sim \hat{t_i}]$
            end for
        end for
    if $|\rho_2| < \lceil \frac{L}{2} \rceil$ then push $(R', \rho_2)$ into $S$
    end for
end while
```

Figure 5.5: Algorithm *Path-based random walk*

In Relom all relevant relations of the database and the probabilities are stored in main memory. This makes the whole process very efficient. For most applications that do not involve too much data, this approach is feasible. Even if the database itself is very large, the data involved in the process of random walk from a certain set of target tuples can usually fit in the main memory.

There is an alternative implementation, in which the database is stored in a backend database server, and only the random walk probabilities are stored in main memory. Because the training set contains a small number of target tuples (no more than a few thousand), the depth of search is limited ($\lceil \frac{L}{2} \rceil = 4$ in our experiments), and joins that have very high cardinality are automatically blocked (e.g., *Conference* $\to$ *Publication* in DBLP database), the total number of entries for random walk probabilities is reasonably small and can be easily stored in main memory even for very large databases. In the process of propagating probabilities, SQL queries are used to retrieve tuples joinable with certain tuples, or cardinalities of such joins. For example, if Relom needs to find all tuples in *Advise* relation that are joinable with tuple "Jiawei Han" in *Professor* relation, the following SQL query is executed:

```
SELECT * FROM Advise WHERE professor = 'Jiawei Han'
```

After computing the probabilities as above, Relom can easily compute the random walk probability between a pair of target tuples following Equation (5.2). It first identifies all qualified pairs of join paths ρ_1 and ρ_2 as in Equation (5.2). Then for each pair of join paths that meet at relation R, it retrieves all probabilities of tuples in R along ρ_1 and ρ_2, and computes the random walk probability between each pair of target tuples along join path $\rho_1 \bowtie \overleftarrow{\rho_2}$. This can be done in a highly efficient way since all probabilities are stored in hashtables and can be accessed in constant time. Our experiments show that Relom can run efficiently even for large databases such as DBLP.

5.4.3 Handling Shared Attribute Values

So far we have discussed how to utilize the paths between different target tuples for detecting duplicates. However, there is another type of important information that has been missing. As shown in Figure 5.6, if two target tuples share common value on a certain attribute in some relation, it indicates some relationship between them, which may play an important role in duplicate detection.

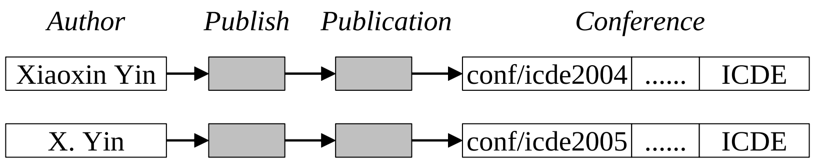


Figure 5.6: Shared attribute values of tuples

Relom utilizes such information in the following way. In principle, the attribute value of a tuple (except primary key) is considered as another tuple. For example, in the example in Figure 5.6, "ICDE" is considered as an individual tuple, which is joinable with tuples "conf/icde2004" and "conf/icde2005". Thus "Xiaoxin Yin" and "X. Yin" can be connected by path "Xiaoxin Yin $\rightarrow \cdots \rightarrow$ conf/icde2004 $\rightarrow$ ICDE $\rightarrow$ conf/icde2005 $\rightarrow \cdots \rightarrow$ X. Yin". In this way we can use a single model to utilize both neighbor tuples and their attribute values.

5.4.4 Building Model of Path-based Random Walk

We have presented our model of path-based random walk and our methods for efficiently computing random walk probabilities. In this section we will discuss how to build such a model. Since the join paths are already determined given the database schema, the major task for building such a model is to determine the weight of each join path.

Relom uses a training set that contains pairs of duplicate tuples (positive examples) and pairs of distinct tuples (negative examples). Such a training set can be created by users by manual labelling, which might be very tedious and time-consuming. We create such training sets automatically. A negative example is created by randomly selecting a pair of tuples. We use certain heuristics to make sure that such pairs of tuples are distinct, such as the edit distance of their names is above a certain threshold.

In order to create a positive example, we randomly select a tuple $\hat{t}$, and split all tuples containing foreign-keys pointing to $\hat{t}$ into two parts. For example, for a student $\hat{t}$ in CS Dept database, we first create a duplicate entry of student $\hat{t'}$, whose name is similar to that of $\hat{t}$. Then we find all tuples in relations *Advise*, *Work-In*, *Publish*, and *Register* that points to $\hat{t}$. For each tuple found above, with probability 0.5, we modify it and let it point to $\hat{t'}$ instead of $\hat{t}$. In this way, we have created a pair of duplicate students.

Suppose a training set is given, and $\mathcal{P}$ is the set of all join paths that both start and end at the target relation with length no greater than L. For each example that consists of a pair of tuples $\hat{t_1}$ and $\hat{t_2}$, Relom computes random walk probabilities between $\hat{t_1}$ and $\hat{t_2}$ along each join path: $P_\rho[\hat{t_1} \sim \hat{t_2}]$ and $P_\rho[\hat{t_2} \sim \hat{t_1}]$, and uses $P_\rho[\hat{t_1} \sim \hat{t_2}] + P_\rho[\hat{t_2} \sim \hat{t_1}]$ as similarity between $\hat{t_1}$ and $\hat{t_2}$ w.r.t. ρ. A vector is created for each example, which contains the similarity of the two tuples w.r.t. each join path in $\mathcal{P}$.

In order to determine the weight of each join path, Relom uses support vector machines [19][1], the state-of-the-art approach of classification. Linear kernel is used for SVM, which produces linear classifiers that are linear combinations of different features (join paths in our case). Thus the weight of each join path can be easily determined from the model built by SVM.

After building the model of path-based random walk, Relom can judge whether a pair of tuples are duplicates or not, thus can identify duplicate tuples in the testing dataset. Because it is often impossible to compute the connectivity between each pair of target tuples in a large dataset (e.g., DBLP has more than 300,000 authors), Relom relies on some inexpensive method to find all candidate pairs of tuples (e.g., pairs of authors whose names have edit-distance less than a certain threshold). There have been many studies for efficiently finding such candidates [21, 52, 63, 76], which are either based on q-grams and inverted index, or search trees in metric spaces. Since finding candidates and duplicate detection are two independent procedures, any method above can be used in Relom. In this chapter we use a naive method that computes the edit distance between names of each pair of target tuples (e.g., authors). This usually leads to a large set of candidates.

[1] Relom uses SVM-*light* at http://svmlight.joachims.org

Relom computes the random walk probability between each pair of tuples using the model of path-based random walk. In most tasks of duplicate detection, negative examples greatly outnumbers positive examples. For example, searching for duplicates in tens of thousands of authors in DBLP is like searching for a needle in a haystack. Therefore, we use precision-recall curves instead of accuracy to evaluate the performance of duplicate detection.

5.5 Empirical Study

In this section we report our experimental results that validate the accuracy and scalability of Relom. Relom is implemented by Microsoft Visual Studio.Net. All experiments are performed on a 2.4GHz Intel PC with 1GB memory, running Windows XP Professional. As mentioned before, we set L (maximum length of paths) to 8.

We compare Relom with Marlin[11], a recent approach representing the state of the art for record linkage with supervised learning approaches. Because there has not been mature approaches for multi-relational record linkage, it is inevitable for us to choose a single-table record linkage approach and adapt it to utilize multi-relational information. We do not choose probabilistic approaches such as [42] which are difficult to be adapted to set-valued attributes, or string similarity based approaches such as [4] and [21] which do not use training data. We choose Marlin because it is easier to be adapted to handle multi-relational data. In a task of duplicate detection, we can compress the relational database into a table, by using each attribute in each relation joinable with the target relation as an attribute in the table. For example, for a join path *Author* → *Publish* → *Publication* in DBLP database, every attribute in *Publication* is used as an attribute in the table, by collapsing values of all tuples joinable with each author into a bag of values. For example, if "Xiaoxin Yin" publishes two papers in 2004 and one in 2003, then he will have value "{2004, 2004, 2003}" in the attribute of *Publication.year*. Because Marlin is designed to handle attributes that contain bags of words, it can be applied on such tables with set-valued attributes. This modified version of Marlin uses richer relational information than existing approaches for duplicate detection for relation data [4, 8, 36], and uses SVM [19], the state-of-the-art machine learning approach to build models. Thus we believe it is sufficient to compare Relom with this modified Marlin.

In fact Relom and Marlin use the same information for duplicate detection. They both use relations joinable with the target relation via join paths of length no greater than $\lceil \frac{L}{2} \rceil$. All attributes that are strings (paper titles, conference titles, and course names) are converted into bags of keywords. (Relom treats each keyword as a value of that attribute, while Marlin can handle bags of keywords directly.) All other attributes (including paper keys, people names, etc.) are used as single values. The major difference between Relom and Marlin is that, Marlin considers such information as a single table with many attributes, while Relom considers it as many paths between target tuples, and thus utilizes arbitrary linkages that start and end at the target tuples.

We use two real datasets in our experiments. The first one is the CS Dept dataset [26] shown in Figure 5.1. It is collected from web sources of the Department of Computer Science at UIUC, in August 2003. All relations contain real data, except *Register* relation which is randomly generated and is not considered in our experiments. The target relation is *Student*.

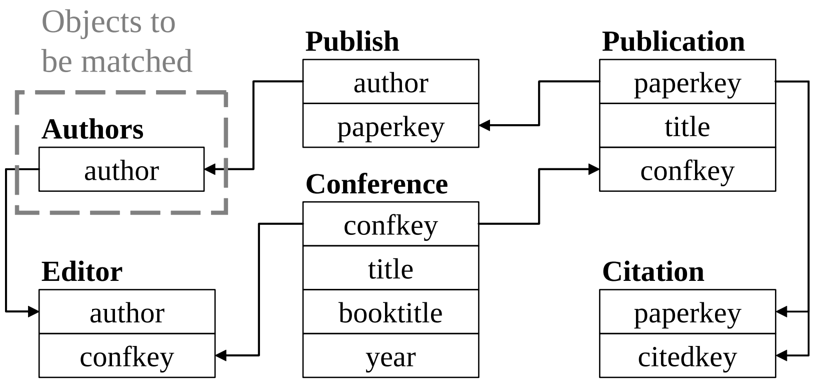


Figure 5.7: Schema of DBLP database

The second dataset is the DBLP dataset [32] shown in Figure 5.7. It is converted from the XML data from DBLP web site. Instead of using the up-to-date data that changes everyday, we use an older version that does not change over time, which is the file "dblp20040213.xml.gz" from http://dblp.uni-trier.de/xml/. It contains information about 316076 authors, 452397 papers, and 1029708 tuples of authorship. The target relation is *Author*.

5.5.1 CS Dept Dataset

The CS Dept dataset does not contain duplicate students, thus we need to add duplicates to enable training and testing. Two-fold experiments are used. All 184 students are randomly divided into two sets, and we split each student in one set into two tuples (as described in Section 5.4.4). In this way two datasets are created, each containing 92 original tuples and 92 pairs of duplicate tuples. In two fold experiments, we use one dataset as training set (with duplicates known to us)[2], and build models with Relom and Marlin. The

[2]When creating a training set in CS Dept dataset, it is a little tricky to create negative examples. Since a positive example is created by splitting all tuples containing foreign-keys pointing to a student, the two students in a positive example will not share common values on attributes involving advisors and research groups (each student usually has one advisor and works in

other dataset is used as test set. We use edit-distance between student names to find all candidate pairs of tuples in the test set, and use Relom and Marlin to judge for each candidate pair whether they are duplicates or not.

The distance between two names is defined as a weighted average of distances between last names, first names, and middle names. The weights of the three parts are: last name 1/2, first name 1/3, and middle name 1/6. The distance between two last names ln_1 and ln_2 is defined as $\frac{6 \times edit\_distance(ln_1, ln_2)}{ln_1.length + ln_2.length}$. The distance between two first names or middle names is defined in the same way, except that the distance between a name and its initial (e.g. 'Steven' and 'S.') is set to 0.3. Two names are considered as candidates for duplicates if their distance is less than a threshold (0.7 in CS Dept dataset).

There are 705 candidate pairs of students found in the two folds, among which 184 are positive. Both Relom and Marlin rank all candidates using their models. Three different SVM kernels (linear, polynomial and Gaussian) are used in Marlin. Relom only uses linear kernel. The recall-precision curves of both approaches are shown in Figure 5.8. The CS Dept dataset is quite small and can be put in main memory. All approaches finish in less than 2 seconds.

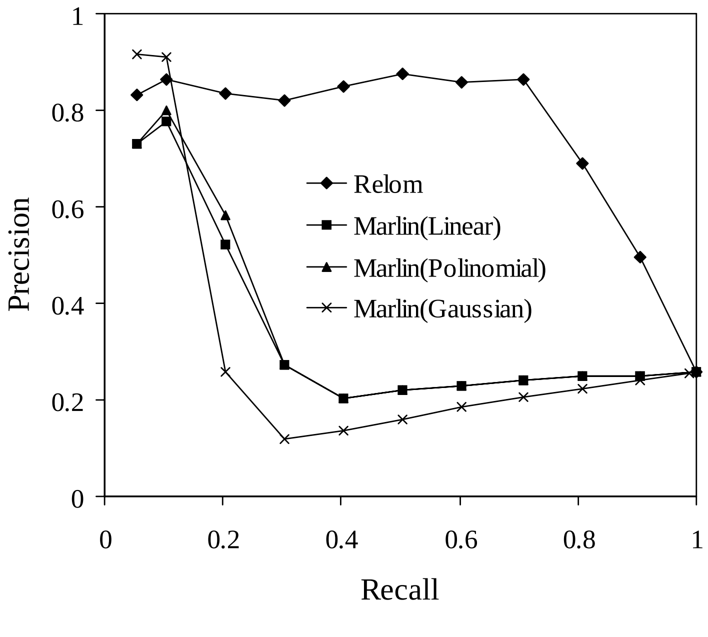


Figure 5.8: Accuracy on CS Dept dataset

one group). If negative examples are created with a pair of original students, they usually share common values on attributes such as years of advisor's publications, or position of advisor (e.g., professor). Thus it is trivial to distinguish positive and negative examples, because negative examples usually have non-zero similarity on certain attributes, while positive examples usually have zero similarity on them. In order to make this dataset meaningful, we "split" negative examples in the same way as we split positive ones. For each student used in a negative example, we only keep half foreign-keys pointing to it, and the two students in a negative example cannot both have values in relations *Advisor* and *Work-In*. This is fair for both Relom and Marlin.

One can see that although both Relom and Marlin achieve high precision at low recalls, Relom achieves much higher precision when recall is higher. This is because there are a small portion of duplicate students that can be identified using shared values in the table created for Marlin by collapsing relational data (e.g., co-authors, conferences of publications). However, many other duplicate students can only be detected using linkages between tuples. For example, a student usually works in the same group with her advisor, co-authors papers with her advisor and group mates, etc. Relom can easily find such duplicates, but Marlin cannot.

5.5.2 DBLP Dataset

From our experiences on DBLP dataset, we believe that it has at least a few hundred duplicate authors. However, nobody knows the exact set of duplicate authors, and in this section we still add duplication into it in order to test the precision and recall of different approaches. In next section we will present results of Relom on the original dataset, which are real duplicates in DBLP.

In this experiment we only consider authors with at least 3 papers (there are 79957 of them) for the following reasons: (1) If an author has only one or two papers, it is often hard even for human to tell whether she is the duplicate of another author, (2) 87% of papers remain in the dataset after unproductive authors are ignored.

In order to create a duplicate author, we randomly find an author $\hat{t}$ of at least 10 papers, modify $\hat{t}$'s name slightly to create a new author $\hat{t}'$. For each tuple containing a foreign-key pointing to $\hat{t}$, we redirect it to $\hat{t}'$ with probability 0.5. We create 4000 duplicate authors in DBLP dataset, which is much more than the original duplicates in DBLP according to our experiences.

Relom and Marlin use this dataset both for training and testing. They create a certain number of (1000 in this experiment) positive examples by splitting tuples pointing to authors, and create the same number of negative examples by randomly selecting authors with completely different names. After building models from training sets, they find candidates of duplicates, which are pairs of names with distance less than 0.3 (as defined in Section 5.5.1). Two different forms of a name (e.g. "S. G. Parker" and "Steven Parker") will definitely be considered as a candidate pair. 15356 candidate pairs are found, among which the 4000 synthetic duplicates are considered as positive.

The recall-precision curves of Relom and Marlin are shown in Figure 5.9. (SVM with Polynomial kernel fails to finish in reasonable time.) It can be seen that both approaches are accurate in identifying duplicates, and the precision of Relom is 4-5% higher than that of Marlin at most recalls.

We leave the judgement of whether this improvement is significant to the readers, by providing the relevant facts. On one hand, the join path *Author* → *Publish* → *Publication* plays the dominant role in this

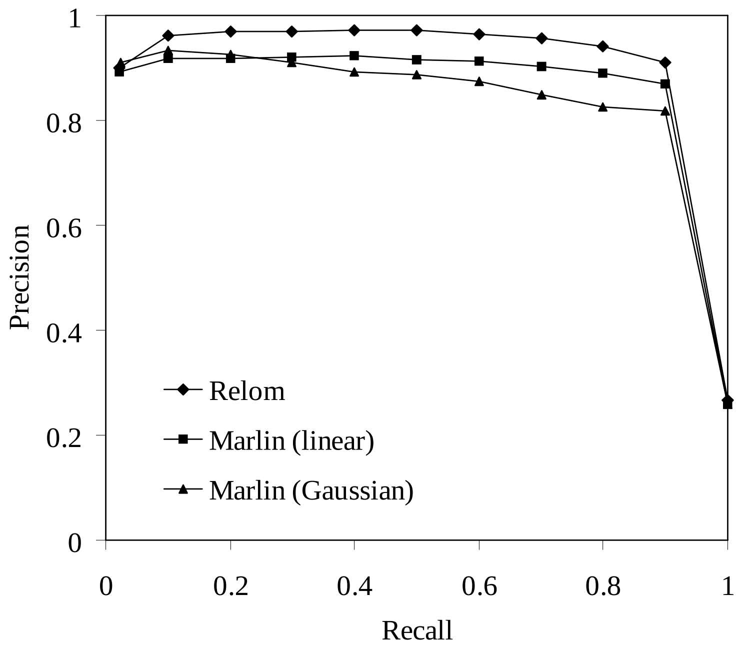


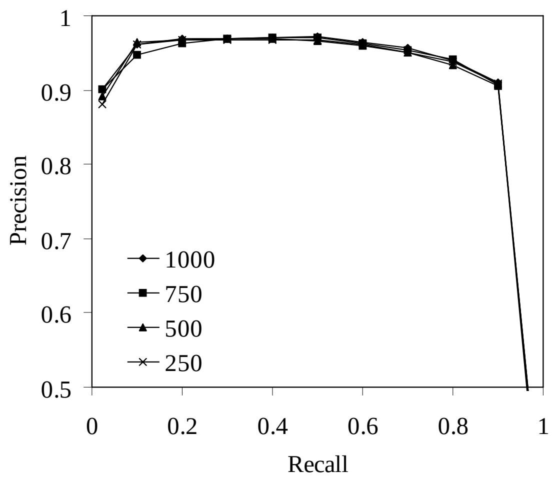
Figure 5.9: Accuracy on DBLP dataset

experiment, since *Publish* relation has 756990 tuples, while relations on alternative join paths have less than 100000 tuples. On the other hand, all duplicate authors are created from the 19989 authors with at least 10 papers. 4041 authors among them have been editors, and 5960 of them have papers cited in *Citation* relation. In our opinion, the improvement of 4-5% is significant, and is also a reasonable result.

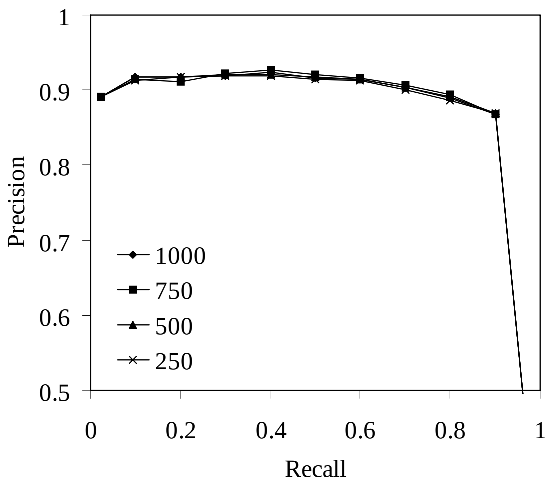
We use 1000 positive and 1000 negative examples in above experiments. We do not use larger training sets because the training set size has little influences on accuracy. We test the accuracy of Relom and Marlin (linear kernel) on training sets of different sizes (from 250 to 1000 positive and negative examples), and show the results in Figure 5.10. The recall-precision curves of different training sets are very close to each other. Thus we do not think a larger training set will improve accuracy significantly.

5.5.3 Detecting Real Duplicates in DBLP

In this section we present the result of Relom on the original DBLP dataset with no synthetic duplicates. Relom creates a training set with 1000 positive and 1000 negative examples, builds a model, and uses it to rank all candidates of duplicates. For the top 200 candidates identified by Relom, we manually judge whether each of them is a real pair of duplicates of not, with the following information. (1) If we can find the web site of an author, we can judge whether the two names refer to the same author. (2) If we can find papers (in PDF or PS) on the web, or abstract pages on *ACM Portal* or *IEEEXplore*, we can usually find



a) Relom



b) Marlin

Figure 5.10: Accuracy with different numbers of training examples

affiliations of authors at certain years, which can help to judge whether two names are duplicates. (3) In the rare case that all above information is unavailable, we judge by their co-authors.

There are four cases for each candidate pair of names. (1) *Match*: all (or most) papers of one name are written by the same author who wrote all (or most) papers of the other name. (2) *Partial match*: some papers of the two names are written by the same author. (3) *Non-match*: no papers of one name are written by authors of papers of the other name. (4) *Undecided*: there is not sufficient information to make a judgement (e.g., for authors in early years).

	match	*partial*	*non-match*	*undecided*
Number	150	34	10	6
Proportion	75%	17%	5%	3%

Table 5.1: Real duplicates in DBLP

The results are shown in Table 5.1. Relom achieves the high accuracy ($\frac{matched+partial}{matched+partial+nonmatch}$) of 94.8%. It is a very tedious and time-consuming job to manually identify whether two names are duplicates, and thus we do not perform the same task for Marlin. We put the top 200 duplicate names and their publications at http://www.ews.uiuc.edu/∼xyin1/projects/DBLP_duplicates.html, together with our evidences for each judgement. The reader is welcomed to inspect our results and notify us about any mistakes we have made.

There are mainly two types of duplicates in DBLP. (1) *Different forms of names*: For example, "Jan M. Smith" and "Jan Smith" both refer to a professor Sweden. Some asian names may be written in different ways, such as “Hong-Jiang Zhang” and “Hongjiang Zhang”. (2) *Typos*, such as “Zhengqiang Chen” and “Zhenqiang Chen”, or “Constantin Halatsis” and “Constantine Halatsis”. We found another interesting case that does belong to the above two categories. “Alon Y. Levy” and “Alon Y. Halevy” are names of the same professor in different periods of time. These two entries have high score of duplication according to Relom (ranked at top 100) because they are highly connected with each other.

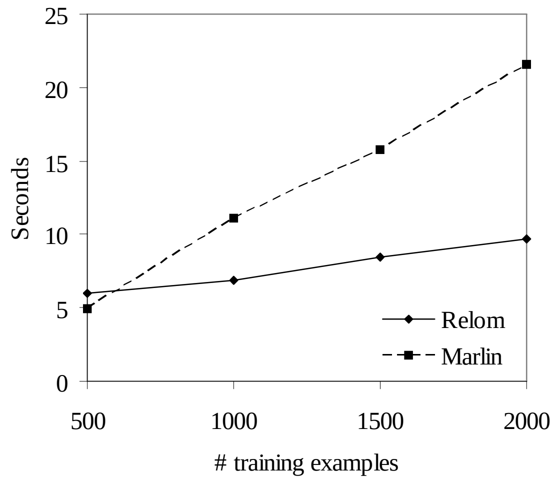
Since we use a stable but older version of DBLP data, some duplicates have been corrected by DBLP. For example, if one searches “Ralph Martin” in DBLP, she will be led to the page of “Ralph R. Martin”, although one can find both entries in the version used by us. Some authors seemed to be deleted when merging duplicates, such as “Georges Gielen” (changed to “Georges G. E. Gielen”). We found 72 of the 184 duplicated authors found by Relom have been merged by DBLP. We do not know how such duplicates are detected, but we think some of them may be reported by the original authors (actually the first author reported once).

5.5.4 Experiments on Scalability

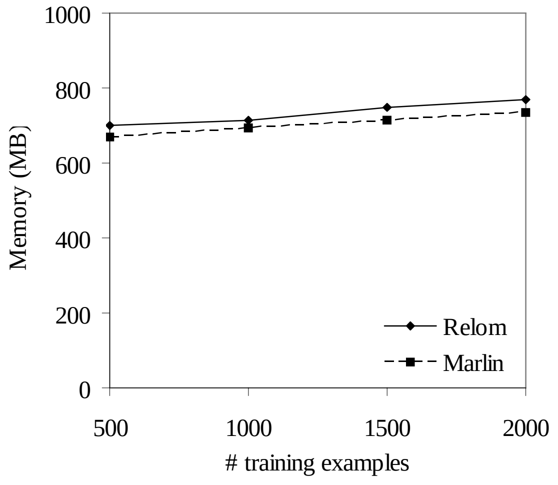
In this section we test the scalability of Relom and Marlin on large datasets. We use the whole DBLP dataset (with all 316076 authors and 452397 papers) in this experiment.

We first test the scalability of Relom w.r.t. training set size. We use the whole DBLP dataset, with from 500 to 2000 training examples (half positive and half negative). The running time and memory usage are shown in Figure 5.11 (a) and (b). One can see that both approaches are linear scalable w.r.t. size of training set. Relom consumes less CPU time, because it propagates probabilities among objects which can be represented by their IDs, while Marlin needs to process the textual information of the tuples. The memory consumption of both approaches are very slightly affected by the training set size, because most memory usage is dedicated to storage of the database. Relom consumes slightly more memory because it needs to store the probabilities.

Then we use different portions of all the authors (from 20% to 100%) to test the scalability of Relom and Marlin with respect to the size of database. When creating a database with $x\%$ of all authors, we put all relevant information of these authors in the database, including their publications, conferences, editorship, and relevant citations. The running time and memory consumption of the two approaches are shown in Figure 5.12 (a) and (b). Training sets of 2000 examples (half positive and half negative) are used. We can see that the running time is not very much affected by the database size, because even when the database is much larger, it still takes a similar amount of time to create a training set of 2000 examples. However, when the database is large, random accesses to some data structures (*e.g.*, sorted trees for indexing) are



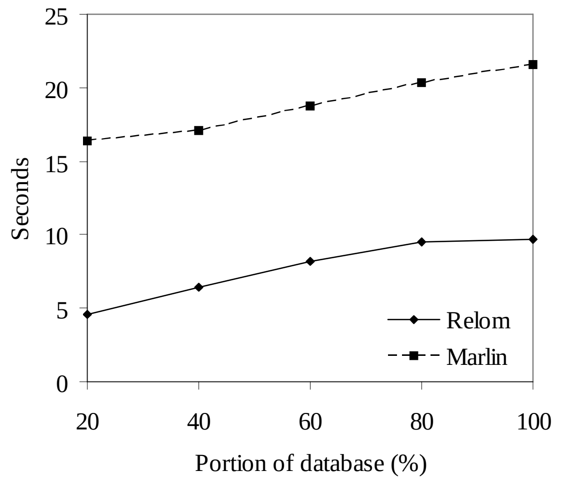
a) Running time



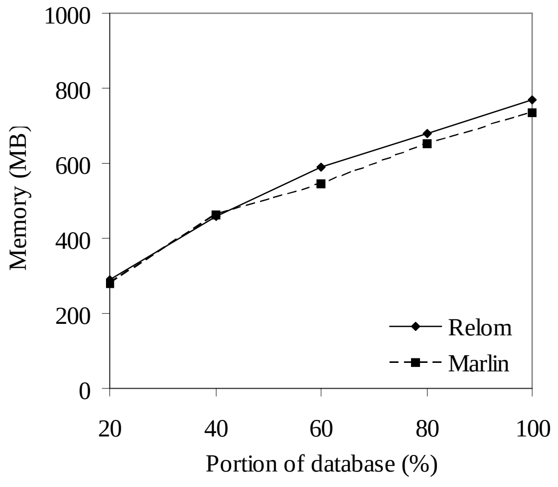
b) Memory usage

Figure 5.11: Scalability w.r.t. training set size

more expensive, and the total running time will be longer. The memory consumption grows sub-linearly with the portion of authors used, as the database size also grows sub-linearly with the portion of authors.



a) Running time



b) Memory usage

Figure 5.12: Scalability w.r.t. database size

5.6 Related Work

The problem of record linkage [107] (also known as deduplication [97] or merge/purge [59]) was originally proposed in [85] and was placed into a statistical framework in [42]. Most subsequent studies are based on computing pairwise similarities and collapsing similar tuples [4, 11, 21, 59, 63]. In [59] an approach was proposed based on sorted neighborhood for efficient record linkage. There were substantial amount of work on using approximate string similarities or TF/IDF model for duplicate detection, such as [21, 25, 52].

In recent years machine learning based approaches [11, 34, 36, 99, 97] received much attention in record linkage. Their common principle is to use a set of training examples (containing pairs of duplicate or distinct tuples) to build a classification model to predict whether a pair of tuples are duplicates or not. Different classifiers such as SVM [19], decision tree [95] and Naive Bayes [77] are used. As the most widely used classifier, SVM is the most popular approach in record linkage, and its high accuracy is verified in [11] and [97] (without active learning). That is why we compare Relom with Marlin [11], the most recent learning-based record linkage approach, which is based on SVM and text vector model.

In [99] an approach based on conditional random field [72] was used to model the influences between the reconciliation decisions of different tuples/values. It builds a graph by representing each pair of tuples/values to be reconciliated as a node, and representing the influence between two reconciliation decisions as an edge. It is shown that such an approach improves accuracy of record linkage. However, the numbers of nodes and edges in the graph are quadratic to the numbers of tuples/values in the database, which is unaffordable for large databases.

In a very recent paper [36] the Semex system is presented, which uses similar ideas as [99] to reconcile references for personal information management by considering the influences between the reconciliation decisions of different tuples/values. Semex uses pruning techniques to reduce the number of nodes and edges in the graph. However, there are still a large number of reconciliation decisions and possible influences between such decisions in a large database, which leads to the generation of large graphs that may not fit in main memory. We do not compare Relom with Semex because: (1) Semex requires domain knowledge on extracting and reconciling data of certain types (e.g., emails and person names), while Relom requires no knowledge from users in duplicate detection; (2) Relom is based on supervised learning, while Semex is based on unsupervised learning.

In [4] an approach is proposed to work on a concept hierarchy (e.g. street address, city, state, and country), and can utilize the parent and children of each tuple to help deduplication. In [8] an approach is proposed to work in relational databases and use the tuples directly joinable with each tuple to help matching them. Although the above two approaches utilize relational information, their approaches can be simulated by converting tuples directly joinable with each target tuple along a certain join path into a set-valued attribute, and tables containing set-valued attributes can be handled by deduplication approaches such as [11] and [97]. In contrast, we propose an approach that can utilize the linkages between different tuples to help matching them, thus utilizing a new form of information that is not used by existing approaches.

One problem in record linkage is how to find all candidates of duplication efficiently (e.g. pairs of persons that the edit-distance between their names is less than a threshold). There have been many studies for efficiently finding such candidates [21, 52, 63, 76], which are either based on q-grams and inverted index, or search trees in metric spaces. Since the detection of candidates and reconciliation of them are two independent procedures, any efficient approach for finding candidates can be used in Relom. Therefore, we do not discuss how to find such candidates in this chapter.

5.7 Summary

Duplication exists in most real world databases and data warehouses. Therefore, duplicate detection (or record linkage) plays an important role in data cleaning. Most existing approaches on duplicate detection are only applicable to data stored in single tables. However, most real world databases contain multiple inter-connected relations.

In this chapter we present Relom, a novel approach for detecting duplicates in relational databases. Unlike existing approaches which rely on common attribute values, Relom employs the paths connecting different tuples for duplicate detection. Thus it can utilize complex linkages between different tuples, which cannot be used by previous approaches even we convert a relational database into a table by a universal join (with limited number of joins). We use a model based on random walk to measure the connectivity between different tuples, and propose an efficient algorithm for computing the probabilities of random walk. Our experiments show that Relom achieves significantly higher accuracy than existing approaches, and is linear scalable w.r.t. dataset size.

This work can be extended to exploit more domain knowledge, such as domain-specific matchers for addresses or names. The method of connection-based duplicate detection can also be integrated into existing data integration systems to provide data cleaning functions using relational information.

Chapter 6

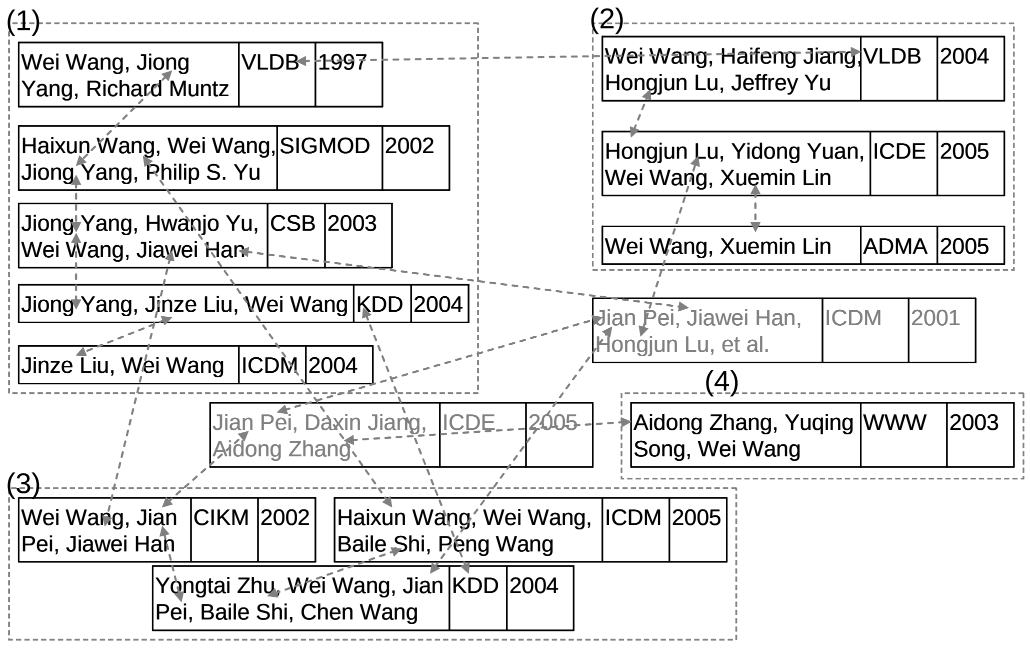
Object Distinction In Relational Databases

In the previous chapter we study the problem of detecting duplicate objects with different names. In this chapter we discuss a similar but different problem, — distinguishing different objects with identical names. Different people or objects may share identical names in the real world, which causes confusion in many applications. For example, a publication database may easily mix up different authors with identical names. It is a nontrivial task to distinguish these objects or people, especially when there is only very limited information associated with each of them. As mentioned in previous chapter (Section 5.3), we found that the linkages among objects provide useful information. Unfortunately, such information is often intertwined and inconsistent with the identities of objects. In this chapter we develop a methodology of object distinction, which utilizes supervised learning to fuse different types of linkages. Our approach combines two complementary measures for relational similarity: *set resemblance of neighbor tuples* and *random walk probability*. Experiments show that our approach can accurately distinguish different objects with identical names in real databases.

6.1 Introduction

People retrieve information from different databases on the Web in their everyday life, such as DBLP, Yahoo shopping, and AllMusic. When querying such databases, one problem that has always been disturbing is that different objects may share identical names. For example, there are 197 papers in DBLP written by at least 14 different "Wei Wang"s. Another example is that there are 72 songs and 3 albums named "Forgotten" in allmusic.com. User are often unable to distinguish them, because the same object may appear in very different contexts, and there is often limited and noisy information associated with each appearance.

In this chapter we study the problem of *Object Distinction, i.e., Distinguishing Objects with Identical Names.* Given a database and a set of references in it referring to multiple objects with identical names, *our goal is to split the references into clusters, so that each cluster corresponds to one real object.* In this chapter we assume the data is stored in a relational database, and the objects to be distinguished and their



(1)
Wei Wang, Jiong Yang, Richard Muntz
VLDB
1997
Haixun Wang, Wei Wang, Jiong Yang, Philip S. Yu
SIGMOD
2002
Jiong Yang, Hwanjo Yu, Wei Wang, Jiawei Han
CSB
2003
Jiong Yang, Jinze Liu, Wei Wang
KDD
2004
Jinze Liu, Wei Wang
ICDM
2004
(2)
Wei Wang, Haifeng Jiang, Hongjun Lu, Jeffrey Yu
VLDB
2004
Hongjun Lu, Yidong Yuan, Wei Wang, Xuemin Lin
ICDE
2005
Wei Wang, Xuemin Lin
ADMA
2005
Jian Pei, Jiawei Han, Hongjun Lu, et al.
ICDM
2001
(4)
Aidong Zhang, Yuqing Song, Wei Wang
WWW
2003
Jian Pei, Daxin Jiang, Aidong Zhang
ICDE
2005
(3)
Wei Wang, Jian Pei, Jiawei Han
CIKM
2002
Haixun Wang, Wei Wang, Baile Shi, Peng Wang
ICDM
2005
Yongtai Zhu, Wei Wang, Jian Pei, Baile Shi, Chen Wang
KDD
2004

(1) Wei Wang at UNC (2) Wei Wang at UNSW, Australia
(3) Wei Wang at Fudan Univ., China (4) Wei Wang at SUNY Buffalo

Figure 6.1: Papers by four different "Wei Wang"s

references reside in a table. A mini example is shown in Figure 6.1, which contains some papers by four different "Wei Wang"s and the linkages among them.

This problem of object distinction is the opposite of a popular problem called *reference reconciliation* (or *record linkage*, *duplicate detection*) [107], which aims at merging records with different contents referring to the same object, such as two citations:

"J. Liu, W. Wang, J. Yang. A framework for ontology driven subspace clustering. KDD 2004."

"Jinze Liu et al. A framework for ontology-driven subspace clustering. SIGKDD Conf. 2004."

referring to the same paper. There have been many record linkage approaches [9, 11, 21, 36, 42, 64, 99]. They usually use some inexpensive techniques [52, 63, 76] to find candidates of duplicate records (*e.g.*, pairs of objects with similar names), and then check whether each pair of candidates are duplicates. Different approaches are used to reconcile each candidate pair, such as probabilistic models of attribute values [36, 42, 99, 107] and textual similarities [11, 21].

Compared with record linkage, objection distinction is a very different problem. First, *because the references have identical names, textual similarity is useless*, *i.e.*, the approaches based on textual similarity [11, 21] cannot be applied. Second, *each reference is usually associated with limited information*, such as the authors, title and venue of a paper in DBLP, and thus it is difficult to make good judgements based on them.

Third and most importantly, *because different references to the same object appear in different contexts, they seldom share common or similar attribute values.* Most record linkage approaches [11, 21, 36, 42, 99, 107] are based on the assumption that duplicate records should have equal or similar values, and thus cannot be used on this problem.

Although the references are associated with limited and possibly inconsistent information, the linkages among references and other objects still provide crucial information for grouping references. For example, in a publication database, different references to authors are connected in numerous ways through authors, conferences and citations. References to the same author are often linked in certain ways, such as through their coauthors, coauthors of coauthors, citations, *etc.*. These linkages provide important information, and a comprehensive analysis on them may likely disclose the identities of objects.

In this chapter we develop a methodology called DISTINCT that can distinguish object identities by fusing different types of linkages with differentiating weights, and using a combination of different similarity measures to assess the value of each linkage. Specifically, we address the following three challenges to achieve high accuracy.

Challenge 1: The linkage information is usually sparse and noisy. Two references to the same object are often unlinked, and all references to the same object may form several weakly linked partitions. On the other hand, there are often linkages between references to different objects.

Example 1 *As shown in Figure 6.1, each "Wei Wang" has different sets of collaborators in different periods of time, because he or she changes affiliation at certain time points. On the other hand, different "Wei Wang"s are often linked. For example, the first and third "Wei Wang"s both coauthored with "Jiawei Han" and "Haixun Wang", and the other two "Wei Wang"s are also linked with them via "Hongjun Lu" or "Aidong Zhang". Moreover, all four "Wei Wang"s work on database and data mining. The intertwined linkage information makes it very difficult to find the identity of each reference even for human.*

From Example 1 one can see that comprehensive analysis on linkages is needed to correctly group references according to their identities. DISTINCT combines two approaches for measuring similarities between records in a relational database. The first approach is *set resemblance between the neighbor tuples* of two records [9] (the *neighbor tuples* of a record are the tuples linked with it). The second is *random walk probability* between two records in the graph of relational data [64]. These two approaches are complementary: one uses the neighborhood information, and the other uses connection strength of linkages.

Challenge 2: How to identify the roles of different types of linkages? There are many types of linkages among references, each following a join path in the database schema. Different types of linkages have very different semantic meanings and different levels of importance. For example, two references being linked by the same coauthor is a much stronger indication that they refer to the same person, in comparison with their linking to the same conference. Existing record linkage approaches for relational databases [9, 64] treat different types of linkages equally, which may bury crucial linkages in many unimportant ones.

DISTINCT uses Support Vector Machines (SVM) [19] to learn a model for weighing different types of linkages. This is the most challenging part of the solution since we want to collect training data without explicit labeling—it is very labor intensive to manually build a training set, especially for non-expert users. Fortunately, although a database contains objects with undistinguishable names, it usually contains many more objects with distinct names, which naturally serves as training data. We design a method for identifying objects with distinct names by analyzing the popularity of object names, and then automatically constructing training examples from those objects. In this way, DISTINCT performs supervised learning without manually labeled training data, which makes it easy to use even for layman users.

Challenge 3: How to group references into clusters accurately and efficiently? As mentioned in Challenge 1, two references to the same object are often unlinked, and there are often linkages between references to different objects. It is insufficient to merely identify pairs of references that refer to the same object, because some identified pairs may be wrong and many true pairs may not be identified. Therefore, it is crucial to use clustering methods to group references based on similarities among them, in order to accurately distinguish references to different objects. Because references to the same object can be merged and considered as a whole, we use agglomerative hierarchical clustering [61], which repeatedly merges the most similar pairs of clusters. However, it is very expensive to repeatedly compute the similarity between each newly created cluster and other existing clusters. We develop an efficient method that computes similarities between clusters incrementally as clusters are merged, and thus avoids redundant computation and achieves high efficiency.

The rest of this chapter is organized as follows. Section 6.2 describes our similarity measure between references. Section 6.3 presents the approach for combining similarities on different join paths using supervised learning. The approach for clustering references is described in Section 6.4, and experimental results are presented in Section 6.5. We discuss related work in Section 6.6 and conclude our study in Section 6.7.

6.2 Similarity Between References

We say a set of references are *resembling* if they have identical textual contents (*e.g.*, references to authors with identical names). Two references are *equivalent* if they refer to the same object, and *distinct* if they do not. Our goal is to group a set of resembling references into clusters so that there is a 1-to-1 mapping between the clusters and the real objects.

In this section we describe our similarity measure for references. Because each reference is usually associated with very limited information, one has to utilize the relationships between each reference and other tuples in the database. We analyze such relationships using the following two types of information: (1) the *neighbor tuples* of each reference and (2) the *linkages* between different references. Based on our observation, for two references, the more overlapping on their neighborhood, or the stronger linkages between them (*e.g.*, short linkages with low fan outs), the more likely they are equivalent.

We introduce a composite similarity measure by adopting and extending two existing similarity measures. The first measure is *set resemblance between neighbor tuples of two references* [9]. We extend it to incorporate the connection strength between each reference and the neighbor tuples. The second measure is *random walk probability between two references within a certain number of steps* [64]. We integrate it with the first measure and compute both measures efficiently using the same framework.

6.2.1 Neighbor Tuples

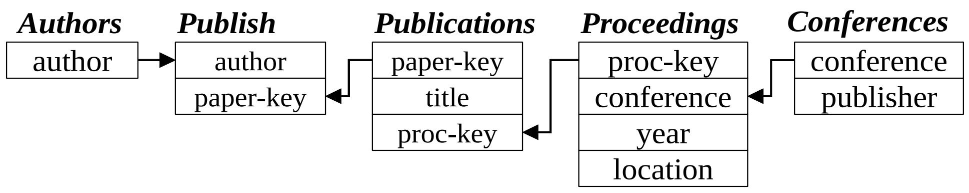


Figure 6.2: The schema of DBLP database

The *neighbor tuples* of a reference are the tuples joinable with it. A reference has a set of neighbor tuples along each join path starting at the relation containing references. The semantic meaning of neighbor tuples is determined by the join path. For example, the schema of DBLP database is shown in Figure 6.2, and we only consider joins between keys and foreign-keys as shown in the figure. We study the references to authors in *Publish* relation (each tuple in *Publish* relation links an author with a paper, and each tuple in *Publication* is usually linked with multiple tuples in *Publish*). The neighbor tuples along join path "$Publish \bowtie Publications \bowtie Publish \bowtie Authors$" represent the authors of the paper for a reference. Because different join paths have very different semantic meanings, we treat the neighbor tuples along different join path separately, and will combine them later by supervised learning.

Definition 14 ***(Neighbor tuples)*** *Suppose the references to be resolved are stored in relation R_r. For a* *reference r that appears in tuple t_r, and a join path P that starts at R_r and ends at relation R_t, the neighbor tuples of r along join path P are the tuples in R_t joinable with t_r along P. We use $NB_P(r)$ to represent the neighbor tuples of reference r along join path P.*

Besides neighbor tuples of each reference, the attribute values of neighbor tuples are also very useful for reconciling references. For example, two neighbor tuples in *Conferences* relation sharing the same value on *publisher* attribute indicates some relationship between these two tuples. In DISTINCT, we consider each value of each attribute (except keys and foreign-keys) as an individual tuple. Let us take the *publisher* attribute in *Proceedings* relation as an example. We create a new relation called *Publisher*, and each distinct value of *publisher* attribute (ACM, Springer, *etc.*) is considered as a tuple in this relation. The *publisher* attribute in *Proceedings* relation is considered as a foreign-key referring to those new tuples. In this way we can use a single model to utilize both neighbor tuples and their attribute values.

6.2.2 Connection Strength

For a reference r and a join path P, the strengths of relationships between r and different tuples in $NB_P(r)$ could be very different. For example, suppose a reference to "Wei Wang" is connected to a paper "vldb/wangym97", and then to two other authors of the paper "Jiong Yang" and "Richard Muntz", who are further connected to other papers. If "Richard Muntz" is a productive author and has many more papers than "Jiong Yang", the relationship between "Wei Wang" and "Richard Muntz" is weaker than that for "Wei Wang" and "Jiong Yang". Moreover, if many papers by "Richard Muntz" and "Jiong Yang" are in KDD conference, then "Wei Wang" has strong relationship with KDD compared with other conferences.

We use probability propagation [106] to model the connection strength between a reference r and its neighbor tuples $NB_P(r)$. Initially the tuple containing r has probability 1. At each step, for each tuple t with non-zero probability, we uniformly propagate t's probability to all tuples joinable with t along the join path P.

Similar to some approaches on record linkage using relational data [9, 64], in DISTINCT we consider join paths with length no more than L. For each qualified join path P, we propagate probability from each reference r along P. For each tuple $t \in NB_P(r)$, we compute $Prob_P(r \rightarrow t)$, the probability of reaching t from r via join path P, which is used as the *connection strength* between r and t. We also compute $Prob_P(t \rightarrow r)$, which is the probability of reaching r from t via the reverse join path of P. This probability can greatly facilitate the computation of random walk probability, as explained in Section 6.2.4.

The computation of both types of probabilities can be done in a depth-first traversal of all qualified join paths. Figure 6.3 shows the procedure of propagating probabilities from a tuple in R_r to tuples in R_1 and

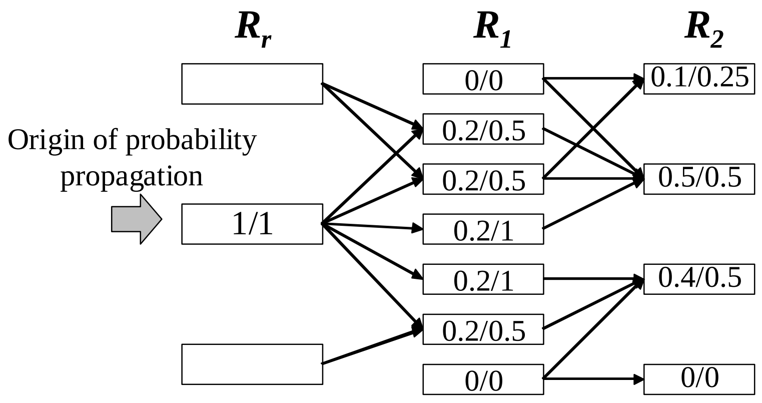


Figure 6.3: Propagating probabilities between tuples

R_2. The two numbers in each box are the probability of reaching this tuple and the probability of reaching the origin from this tuple. Let P_1 be the current join path from R_r to R_1, and P_2 be the join path from R_r to R_2. Let $T_{R_k \bowtie R_l}(t)$ be the set of tuples in R_l joinable with a tuple t in R_k. For each tuple $t_i \in R_1$ we have already computed $Prob_{P_1}(r \to t_i)$ and $Prob_{P_1}(t_i \to r)$. Then for each tuple $t_j \in R_2$, we can compute $Prob_{P_2}(r \to t_j)$ according to the definition:

$$Prob_{P_2}(r \to t_j) = \sum_{t_i \in T_{R_2 \bowtie R_1}(t_j)} \frac{Prob_{P_1}(r \to t_i)}{|T_{R_1 \bowtie R_2}(t_i)|} \tag{6.1}$$

$Prob_{P_2}(t_j \to r)$ can be computed by

$$Prob_{P_2}(t_j \to r) = \sum_{t_i \in T_{R_2 \bowtie R_1}(t_j)} \frac{Prob_{P_1}(t_i \to r)}{|T_{R_2 \bowtie R_1}(t_j)|} \tag{6.2}$$

6.2.3 Set Resemblance of Neighbor Tuples

Our first measure for similarity between two references r_1 and r_2 is the set resemblance between their neighbor tuples. This measure represents the similarity between the contexts of two references in a relational database. In DBLP database, it indicates whether two references to authors are connected with similar coauthors, publications, conferences, etc. The set resemblance between neighbor tuples is defined by Jaccard coefficient [101]. Because the relationships between a reference and different neighbor tuples have different strength, the connection strengths between references and tuples are incorporated into our definition.

Definition 15 ***(Set Resemblance.)*** *The set resemblance similarity between two references r_1 and r_2 with* *respect to join path P is defined as*

$$Resem_P(r_1, r_2) = \frac{\sum_{t \in NB_P(r_1) \cap NB_P(r_2)} \min(Prob_P(r_1 \to t), Prob_P(r_2 \to t))}{\sum_{t \in NB_P(r_1) \cup NB_P(r_2)} \max(Prob_P(r_1 \to t), Prob_P(r_2 \to t))} \tag{6.3}$$

($Prob_P(r \to t) = 0$ if t is not in $NB_P(r)$.)

DISTINCT stores the neighbor tuples of each reference along each join path in a hashtable. Thus $Resem_P(r_1, r_2)$ can be computed in linear time by scanning the two hashtables.

6.2.4 Random Walk Probability

Set resemblance similarity represents the similarity between the contexts of references. There is another important factor for similarity, which are the linkages between them. Different linkages have very different strengths. For example, consider two references to authors r_1 and r_2. If there is a linkage between r_1 and r_2 through "ICDE 2006", then this linkage is pretty weak because "ICDE 2006" is connected to many authors. If r_1 and r_2 both coauthor with author a_3 for many times, then the linkage between r_1 and r_2 through a_3 is quite strong.

Based on the above analysis, we adopt the random walk probability model used in multi-relational record linkage [64]. It performs probability propagation as in Section 6.2.2, and the total strength of the linkages between two references is defined as the probability of walking from one reference to the other within a certain number of steps. A novel feature of our approach is that, we compute the random walk probability along each join path separately, in order to acknowledge the different semantics of different join paths. Machine learning techniques will be used to combine the random walk probabilities through different join paths, which is described in Section 6.3.

It is very expensive to compute random walk probabilities along long join paths. On the other hand, since we have computed the probabilities of walking from references to their neighbor tuples, and those from neighbor tuples to references, we can easily compute the probability of walking between two references by combining such probabilities. Let $\mathcal{P}$ be the set of join paths no longer than L starting from the relation containing references. For each join path $P \in \mathcal{P}$, for a reference r and each tuple $t \in NB_P(r)$, we have computed the probability of walking from r to t and that from t to r. Because each join path no longer than $2L$ can be decomposed into two join paths no longer than L, we can compute the random walk probabilities between two references through join paths no longer than $2L$ by combining probabilities along join paths no longer than L.

Lemma 5 *Let $\overleftarrow{P}$ represent the reverse join path of join path P. For each join path Q so that $|Q| \leq 2L$, Q*

can be decomposed as $Q = P_1 \bowtie \overleftarrow{P_2}$, so that $P_1 \in \mathcal{P}$ and $P_2 \in \mathcal{P}$. The random walk probability of walking from reference r_1 to r_2 through Q is

$$Prob_Q(r_1 \to r_2) = \sum_{t \in NB_{P_1}(r_1) \cap NB_{P_2}(r_2)} Prob_{P_1}(r_1 \to t) \times Prob_{P_2}(t \to r_2) \tag{6.4}$$

PROOF. *We can divide the join path Q into two parts P_1 and P_2', so that $|P_1| = \lceil \frac{|Q|}{2} \rceil$ and $|P_2'| = \lfloor \frac{|Q|}{2} \rfloor$. Let $P_2 = \overleftarrow{P_2'}$. If $|Q| \leq 2L$, then $|P_1| \leq L$ and $|P_2| \leq L$, and both P_1 and P_2 start at the relation containing references R_r and end at a relation R_m. According to the definition of random walk, the probability of walking from r_1 to r_2 via Q is the sum of the probability of walking from r_1 to a tuple t in R_m (via P_1) times the probability of walking from t to r_2 (via P_2').* ■

For two references r_1 and r_2, their similarity based on random walk probability through join path Q is defined as the geometric average of the probability of going from r_1 to r_2 and that from r_2 to r_1.

$$WalkProb_Q(r_1, r_2) = \sqrt{Prob_Q(r_1 \to r_2) \cdot Prob_Q(r_2 \to r_1)} \tag{6.5}$$

Geometric average is used because these two probabilities often differ by hundreds or thousands of times, and arithmetic average will ignore the smaller one.

In general, random walk probability indicates how likely it is to walk from one reference to another through all linkages between them. It is a complementary measure to set resemblance, which indicates the context similarity of references. DISTINCT combines both measures to perform comprehensive analysis on similarities between references.

6.3 Supervised Learning with Automatically Constructed Training Set

In record linkages approaches that utilize relation information [9, 64], all join paths are treated equally. However, linkages along different join paths have very different semantic meanings, and thus should have different weights on importance. For example, in DBLP database two references to authors being linked by the same coauthor is a strong indication of possible equivalence, whereas two references being linked by the same conference is much weaker. The schema of a real database is often more complicated than that of DBLP, thus making it more crucial to distinguish important join paths from unimportant ones.

DISTINCT uses supervised learning to determine the pertinence of each join path and assign a weight to it.

Supervised learning has proven effective in record linkage in a single table [11]. But its usage in reconciling records or references in relational databases has not been explored much.

In order to use supervised learning, we need a training set which contains equivalent references as positive examples and distinct references as negative ones. It is labor intensive to manually create a training set, especially for non-expert users. Thus it is highly desirable if one can construct a training set automatically. An important observation is that although a database contains undistinguishable objects, it usually contains many more objects with distinct names. For example, most authors in DBLP (or most songs, movies in entertainment websites) have distinct names, and we can create many training examples from them. We can use a pair of references to an object with unique name as a positive example, and use a pair of references to two different objects as a negative example.

In order to determine whether an object's name is unique, we consider whether each term in the name is frequent. For example, a person's name usually consists of a first name and a last name. We say a last (or first) name is rare if it co-appears with only one or a few first (or last) names. If a person's name contains a rare first name and a rare last name, the name is very likely to be a unique one. In this way we can find many names from the DBLP that are very likely to be unique, and use them to construct training sets. When distinguishing other objects such as songs and movies, we can also utilize the frequencies of words in their names. A name that contains several rare words is very likely to be unique.

Given the training examples, we use Support Vector Machines (SVM) [19] to learn a model based on similarities via different join paths. As shown in previous studies on record linkage [11], SVM is an effective approach for combining different attributes for reconciling records. However, we are using SVM for weighting or differentiating join paths.

We first introduce the learning procedure for set resemblance similarities. Each training example (which is a pair of references) is converted into a vector, and each dimension of the vector represents set resemblance similarity between the two references along a certain join path. Then SVM with linear kernel is applied to these training sets, and the learned model is a linear combination of similarities via different join paths. Usually, some important join paths have high positive weights, whereas others have weights close to zero and can be ignored in further computation.

Let $Resem(r_1, r_2)$ be the overall set resemblance similarity between r_1 and r_2,

$$Resem(r_1, r_2) = \sum_{P \in \mathcal{P}} w(P) \cdot Resem_P(r_1, r_2), \tag{6.6}$$

where $w(P)$ is the weight of join path P.

We apply the same procedure to similarities based on random walk probabilities. Let $WalkProb(r_1, r_2)$

be the overall random walk probability between r_1 and r_2. It is defined in the same way as $Resem(r_1, r_2)$.

6.4 Clustering References

As mentioned earlier, the similarity between references cannot accurately represent their relationships, as two references to the same object are often unlinked, and there are often linkages between references to different objects. Therefore, it is very important to use clustering methods to group resembling references into clusters, so that each cluster corresponds to a real entity. The clustering procedure will be discussed in this section.

6.4.1 Clustering Strategy

There are many clustering approaches [61]. The problem of clustering references has the following features: (1) The references do not lie in an Euclidean space, (2) the number of clusters is completely unknown, and (3) equivalent references can be merged into a cluster, which still represent a single object. Therefore, agglomerative hierarchical clustering is most suitable for this problem.

The agglomerative hierarchical clustering algorithm first uses each reference as a cluster, and then repeatedly merges the most similar pairs of clusters. A most important issue is how to measure the similarity between two clusters of references. There are different measures including Single-Link (highest similarity between two points in two clusters), Complete-Link (lowest similarity between two points in two clusters), and Average-Link (average similarity between all points in two clusters) [61]. As shown in the example of "Wei Wang" in Figure 6.1, references to the same object may form weakly linked partitions, and thus Complete-Link is not appropriate. On the other hand, references to different objects may be linked, which makes Single-Link inappropriate.

In comparison, Average-Link is a reasonable measure, as it captures the overall similarity between two clusters and is not affected by individual linkages which may be misleading. However, it suffers from the following problem. References to the same object often form weakly linked partitions. For example, in DBLP an author may collaborate with different groups of people when she is affiliated with different institutions, which leads to groups of weakly linked references. During the clustering process, some references may be highly related to part of a big cluster, but not related to the other parts. Figure 6.4 shows an example, in which each point represents a reference and points close to each other have high similarity. We can see cluster C_2 is similar to the upper part of cluster C_1, but the average similarity between C_2 and C_1 is pretty low, and they will not be merged using Average-Link.

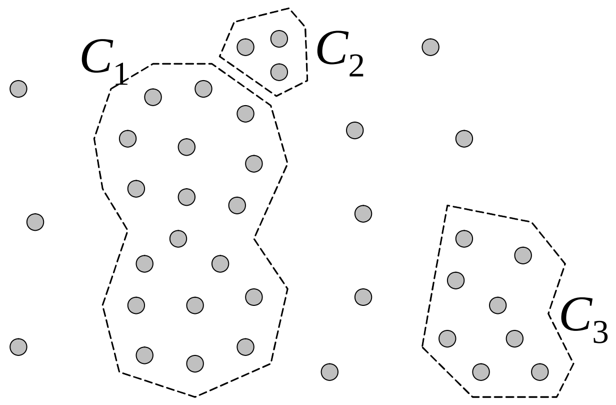


Figure 6.4: Problems with Average-Link

To address this problem, we combine the average similarity with *collective similarity*, which is computed by considering each cluster as a whole. Among our two similarity measures, random walk probability can be more naturally applied to groups of equivalent references, as the random walk probability between two clusters is just the probability of walking from one group of references to another. (The probability of walking from a cluster C_1 to another cluster C_2 is the probability of walking from a uniformly randomly selected reference in C_1 to any reference in C_2.) Using this measure, in Figure 6.4 the average probability of walking between C_1 and C_2 is fairly high, and they will be merged.

Therefore we introduce a composite similarity measure by combining the average set resemblance similarity with the collective random walk probability when measuring similarity between clusters. Because these two measures may have different scales, and arithmetic average will often ignore the smaller one, we use the geometric average of the two measures as the overall similarity between two clusters.

$$Sim(C_1, C_2) = \sqrt{Resem(C_1, C_2) \cdot WalkProb(C_1, C_2)}, \tag{6.7}$$

where $Resem(C_1, C_2)$ is the average set resemblance similarity between references in C_1 and those in C_2, and $WalkProb(C_1, C_2)$ is the collective random walk probability between them.

6.4.2 Efficient Computation

During agglomerative hierarchical clustering, one needs to repeatedly compute similarities between clusters. When clusters are large, it is very expensive to compute the average similarity between two clusters in the brute-force way, and it is also very expensive to compute the collective random walk probability between two clusters as there are often a huge number of linkages between them. Therefore, we design efficient methods that can incrementally compute the similarity between clusters as clusters are merged.

Initially each reference is used as a cluster, and the set resemblance similarity and random walk probability between each pair of clusters are computed. This is usually affordable because the number of references having identical names is seldom very large. At each step, the two most similar clusters C_1 and C_2 are merged into a new cluster C_3, and we need to compute the similarity between C_3 and each existing cluster C_i. When C_3 is very large, a brute-force method may consume similar amount of time as computing pair-wise similarity during initialization, and it is unaffordable to perform such computation at every step.

One important observation for improving efficiency is that, both the average set resemblance similarity and random walk probability between C_3 and C_i can be directly computed by aggregating the similarities between C_1, C_2 and C_i. This is formalized in the following two properties.

Property 1 ***(Additive property of set resemblance.)*** *Let $C_3 = C_1 \cup C_2$. For another cluster C_i,*

$$Resem(C_3, C_i) = \frac{|C_1| \cdot Resem(C_1, C_i) + |C_2| \cdot Resem(C_2, C_i)}{|C_1| + |C_2|}. \tag{6.8}$$

PROOF. *Can be derived from definitions.* ∎

The random walk probability between a merged cluster and existing clusters can also be computed by combining the probabilities involving the two clusters being merged.

Property 2 ***(Additive property of random walk probability.)*** *Let $WalkProb_Q(C_i \rightarrow C_j)$ be the probability of walking from a randomly selected reference in C_i to any reference in C_j through join path Q. If $C_3 = C_1 \cup C_2$, then for another cluster C_i,*

$$WalkProb_Q(C_3 \rightarrow C_i) = \frac{|C_1| \cdot WalkProb_Q(C_1 \rightarrow C_i) + |C_2| \cdot WalkProb_Q(C_2 \rightarrow C_i)}{|C_1| + |C_2|}. \tag{6.9}$$

$$WalkProb_Q(C_i \rightarrow C_3) = WalkProb_Q(C_i \rightarrow C_1) + WalkProb_Q(C_i \rightarrow C_2). \tag{6.10}$$

PROOF. *We first prove Equation (6.9). When randomly selecting a reference r from C_3, the probability of $r \in C_1$ is $\frac{|C_1|}{|C_1|+|C_2|}$, and that of $r \in C_2$ is $\frac{|C_2|}{|C_1|+|C_2|}$. Thus the probability of walking from a random reference in C_3 to cluster C_i is $\frac{|C_1|}{|C_1|+|C_2|} \times WalkProb_Q(C_1 \rightarrow C_i) + \frac{|C_2|}{|C_1|+|C_2|} \times WalkProb_Q(C_2 \rightarrow C_i)$. Then we prove Equation (6.10). The probability of walking from a reference r in C_i to C_3 is the sum of probability of walking from r to C_1 and that of walking from r to C_2.* ∎

The procedure of clustering references is described below.

1. Use each reference as a cluster. Compute the similarity between each pair of clusters. Store the similarities between all pairs of clusters in a binary search tree.

2. Remove the highest similarity. If this similarity is lower than min-sim, go to step 4. Otherwise suppose it is between cluster C_i and C_j. If C_i or C_j has been merged into other clusters and does not exist any more, repeat this step.

3. Create cluster $C_k = C_i \cup C_j$, and remove C_i and C_j. For each existing cluster C_l, compute the average set resemblance similarity and collective random walk probability between C_k and C_l, using Property 1 and 2. Insert the similarity between C_k and C_l into the binary search tree. Go to step 2.

4. Output the clusters.

As shown in [61], the time and space complexity of a regular agglomerative hierarchical clustering algorithm is $O(N^2 \log N)$ and $O(N^2)$. Although we use Average-Link and collective random walk probability, both measures can be aggregated in constant time as shown in Property 1 and 2, and this does not increase complexity. The number of references with identical textual contents is not large in most cases. For example, in DBLP the most productive author has 371 publications, and there are only 580 authors with at least 100 publications. Therefore, our approach can usually cluster references in very short time.

6.5 Experimental Results

In this section we report our empirical study to test the effectiveness and efficiency of the proposed approach for distinguishing references of different objects with identical names. All algorithms are implemented using Microsoft Visual Studio.Net (C#), and all experiments are conducted on an Intel Pentium IV 3.0GHz PC with 1GB memory, running Microsoft Windows XP professional.

As DISTINCT is the first approach for distinguishing references of different identities, we test the advantages of the new methods that we introduce. We compared DISTINCT with the previous approaches that use each single similarity measure between references—set resemblance [9] or random walk probability [64]. We also compare DISTINCT with the approach using both similarity measures but does not use supervised learning and treats all join paths equally.

6.5.1 Experiments on Accuracy

We test DISTINCT on DBLP database, which is converted from the XML data of DBLP. Its schema is shown in Figure 6.2. We remove authors with no more than 2 papers, and there are 127,124 authors left. There are about 616K papers and 1.29M references to authors in *Publish* relation (authorship). In DBLP we focus on distinguishing references to authors with identical names.

Training

We first build a training set using the method in Section 6.3. We first randomly select 1000 *rare names*, so that it is very likely that each of them represents a single author. In DBLP we consider a name as rare if it has a middle name (or middle initial) and a rare last name. We randomly select a pair of references to each author, so as to create 1000 positive examples. We randomly select 1000 pairs of references to different authors as negative examples. Then SVM with linear kernel is applied on this training set to build a model combining different join paths. The whole process takes 62.13 seconds.

Synthetic Groups of Resembling References

DISTINCT is first tested on synthetic groups of references to different objects with identical names, which are created by changing the names of some objects. We first select 1000 authors with rare names not used in the training data, each having at least 5 papers. Then we divide the 1000 authors into many small groups, each having G authors ($G = 2, 4, 6, 8, 10$), and assign a single name for the authors in each group. DISTINCT is applied on all references to authors in each group, and generates clusters of references. The clusters are compared with the standard clustering with G clusters, each corresponding to an author.

We measure the performance of DISTINCT by precision, recall and accuracy, which are defined by considering each pair of equivalent references as an item in the correct answer. Suppose the standard set of clusters is C^*, and the set of clusters by DISTINCT is C. Let TP (true positive) be the number of pairs of references that are in the same cluster in both C^* and C. Let FP (false positive) be the number of pairs of references in the same cluster in C but not in C^*, and FN (false negative) be the number of pairs of references in the same cluster in C^* but not in C. Precision and recall are defined as

$$precision = \frac{TP}{TP + FP}, \qquad recall = \frac{TP}{TP + FN}.$$

Accuracy is defined by Jaccard coefficient [101]

$$accuracy = \frac{TP}{TP + FP + FN}.$$

A high precision means that references to different authors are seldom merged together, and a high recall means that references to the same author are seldom partitioned. Precision is 1 if each cluster contains only one reference, and recall is 1 if all references are put into one cluster.

An important parameter in clustering references is min-sim. During agglomerative hierarchical clustering, two clusters are merged if their similarity is no less than min-sim. Thus min-sim controls the precision and recall of DISTINCT – a smaller min-sim leads to less clusters, higher recall and lower precision. We will test the performance of DISTINCT with different min-sim, in order to analyze the relationship between precision and recall.

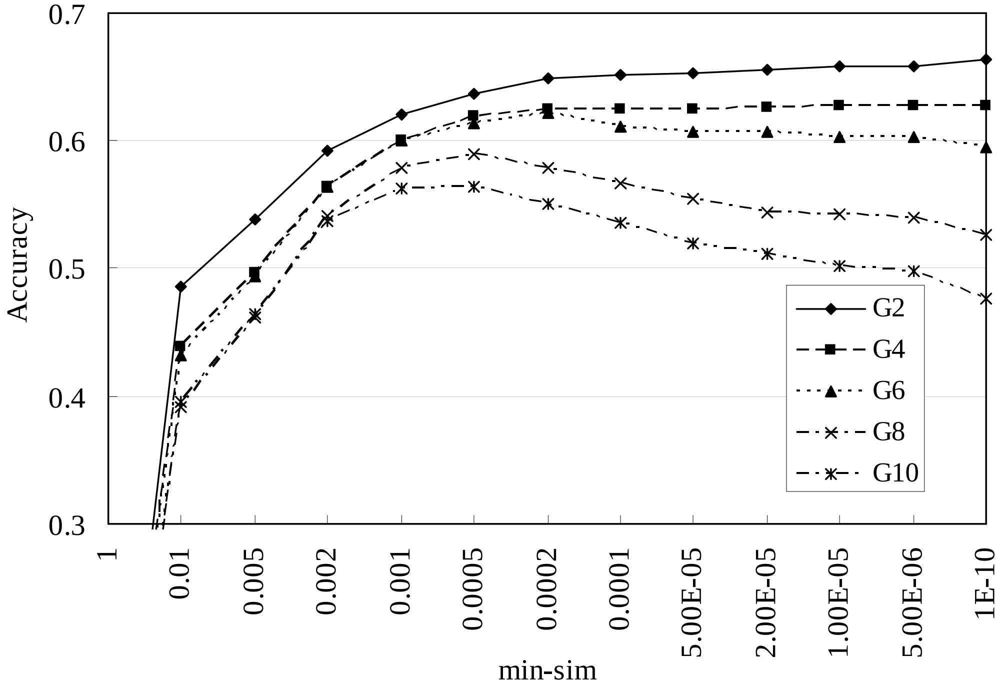


Figure 6.5: Accuracy of DISTINCT with diff. min-sim

The accuracy of DISTINCT on synthetic reference groups to authors with identical names is shown in Figure 6.5, which contains the average accuracy of groups with size 2, 4, 6, 8 and 10 (there are $\lfloor 1000/G \rfloor$ groups with size G). For each group size, we test DISTINCT with 12 min-sim values (starting from 0.01 and decreasing exponentially). Please note that accuracy (Jaccard coefficient) is a tough measure. For two clustering C_1 and C_2, if 80% of equivalent pairs of C_1 and 80% of those of C_2 are overlapped, the accuracy is only $0.8/(0.8 + 0.2 + 0.2) = 0.667$.

The precision-recall curve of DISTINCT is shown in Figure 6.6. One can see that the precision is quite high when group size is small, indicating that references to different authors are seldom put into the same cluster. When group size is large, it can still achieve reasonably high precision and recall (*e.g.*, precision=0.83 and recall=0.65 when each group has 10 authors). This shows that DISTINCT can effectively group references

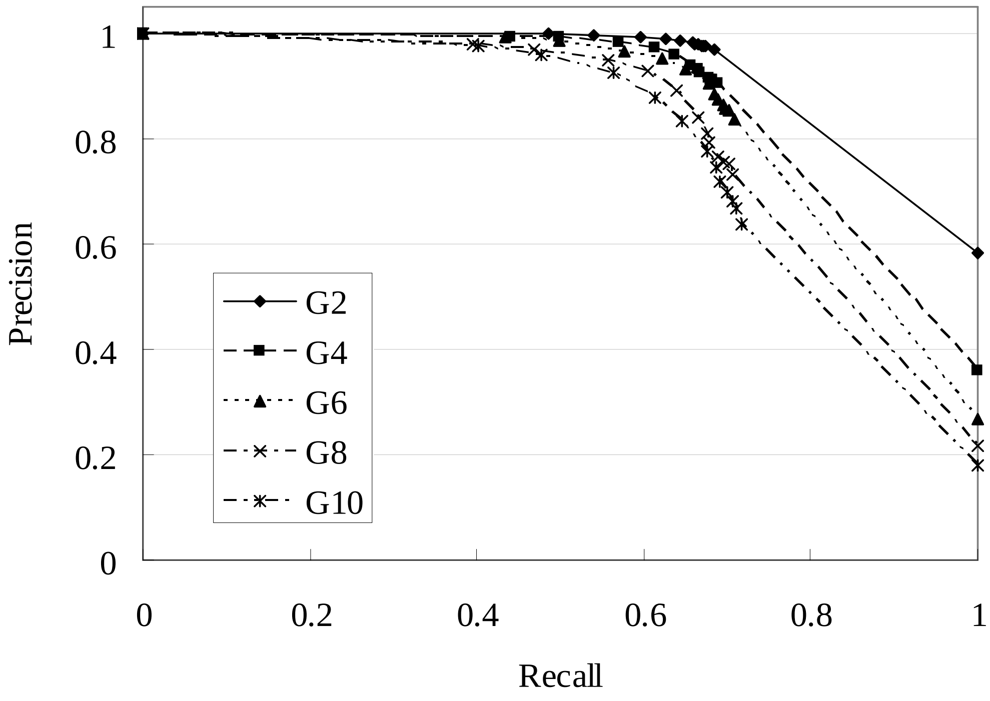
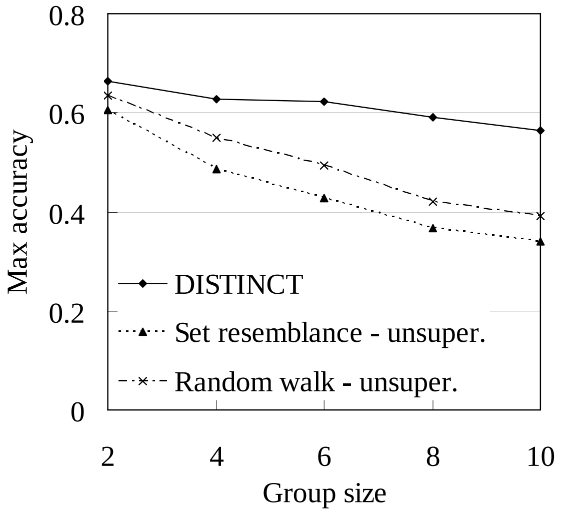
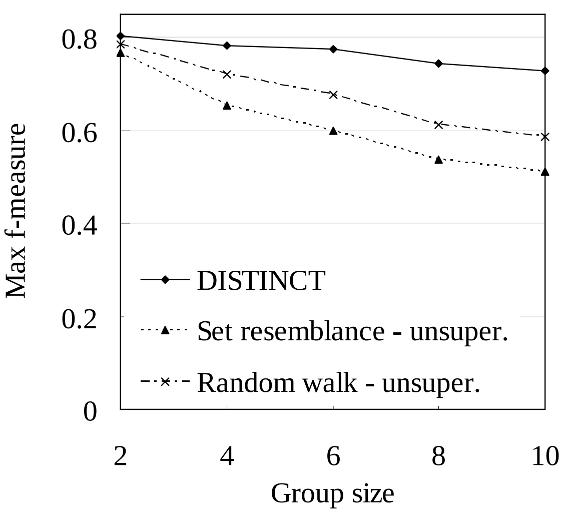


Figure 6.6: Precision-recall curve of DISTINCT

according to their real identities.



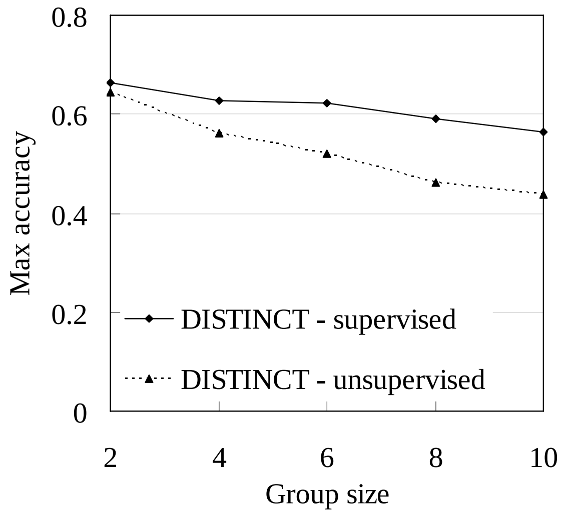
a) Max accuracy



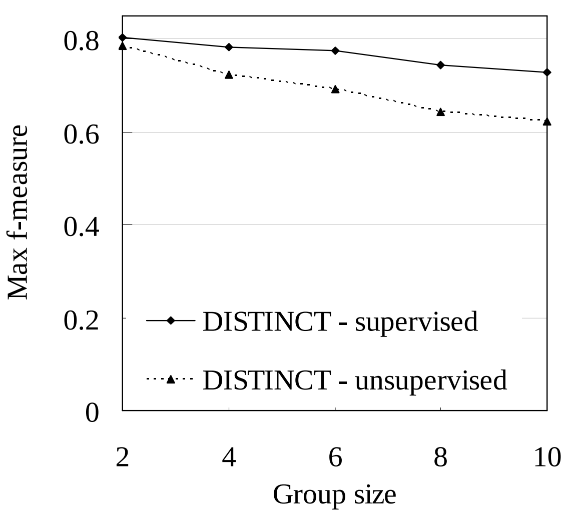
b) Max f-measure

Figure 6.7: DISTINCT v.s. previous approaches

We compare DISTINCT using the composite similarity measure with the existing approaches using each individual similarity measure without supervised learning, including the approach using set-resemblance [9] and that using random walk probability [64]. The same clustering method is used for all three approaches. The highest accuracy and f-measure (using any of the 12 values of min-sim) are shown in Figure 6.7. It can be seen that when group size is 2, DISTINCT improves accuracy and f-measure[1] by only 2-3%. However, the improvement increases sharply with group size, and DISTINCT improves accuracy and f-measure by about 15% when group size is 10. It shows that compared with existing approaches, DISTINCT has more advantages on more difficult problems of distinguishing references.



a) Max accuracy



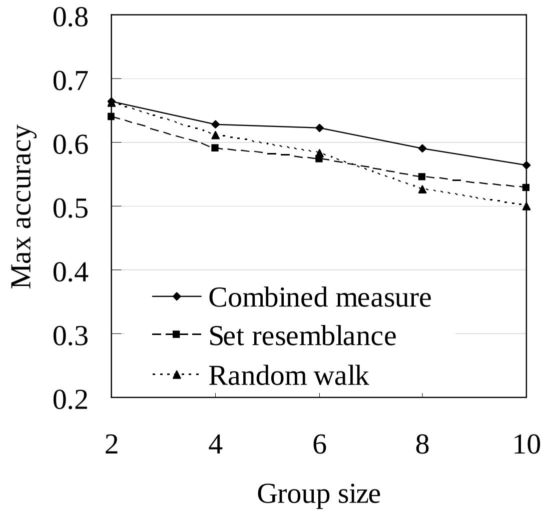
b) Max f-measure

Figure 6.8: DISTINCT with and w/o supervised learning

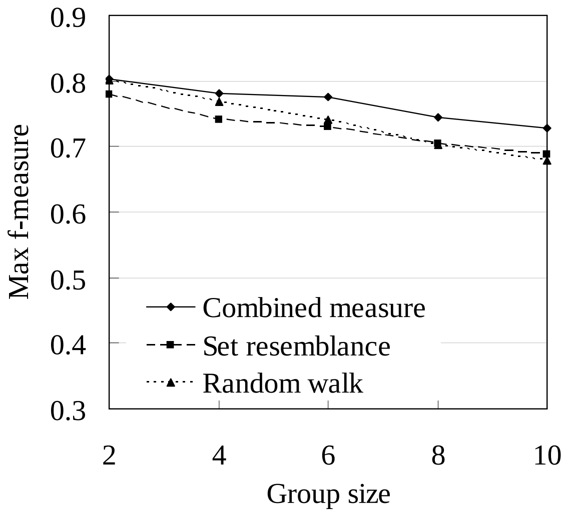
Then we test the effectiveness of supervised learning by comparing DISTINCT with the same approach that does not use supervised learning. The highest accuracy and f-measure are shown in Figure 6.8. One can see that supervised learning improves accuracy and f-measure by only 2% when group size is 2, and improves accuracy by 13% and f-measure by 10% when group size is 10.

We also test the effectiveness of combining two complementary similarity measures, by comparing DISTINCT with the approach that uses each individual measure (with supervised learning). Figure 6.9 shows the highest accuracy and f-measure of the three approaches. One can see that the accuracy is improved by 4% for groups with sizes greater than 6 by combining both measures.

[1]f-measure is the harmonic mean of precision and recall.



a) Max accuracy



b) Max f-measure

Figure 6.9: Combined vs. individual measures

Real Cases

We test DISTINCT on real names in DBLP that correspond to multiple authors. We try many combinations of popular first and last names, and find 10 names corresponding to multiple authors. Table 6.1 shows the 10 names[2], together with the number of authors and number of references.

In order to test the accuracy of our approach, we manually label the references to these names. For each name, we divide the references into groups according to the authors' identities, which are determined by the authors' home pages or affiliations shown on the papers.[3]

Name	#author	#ref
Hui Fang	3	9
Ajay Gupta	4	16
Joseph Hellerstein	2	151
Rakesh Kumar	2	36
Michael Wagner	5	29
Bing Liu	6	89
Jim Smith	3	19
Lei Wang	13	55
Wei Wang	14	141
Bin Yu	5	44

Table 6.1: Names corresponding to multiple authors

[2]We manually merge "Joseph M. Hellerstein" and "Joseph L. Hellerstein" into "Joseph Hellerstein".

[3]References whose author identities cannot be found (*e.g.*, no electronic version of paper) are removed. We also remove authors with only one reference that is not related to other references by coauthors or conferences, because such references will not affect accuracy.

We use DISTINCT to distinguish references to each name. min-sim is set to 0.0005, which leads to highest accuracy in Section 6.5.1 when group size is 8 or 10. Table 6.2 shows the accuracy, precision, and recall of DISTINCT for each name. In general, DISTINCT successfully group references with pretty high accuracy. There is no false positive in 7 out of 10 cases, indicating that DISTINCT seldom merges references to different authors. In the mean time, the average recall is 83.6%, showing that references to the same author can be merged in most cases. In some cases references to one author are divided into multiple groups. For example, 18 references to "Michael Wagner" in Australia are divided into two groups, which leads to low recall.

Name	accuracy	precision	recall	f-measure
Hui Fang	1.0	1.0	1.0	1.0
Ajay Gupta	1.0	1.0	1.0	1.0
Joseph Hellerstein	0.810	1.0	0.810	0.895
Rakesh Kumar	1.0	1.0	1.0	1.0
Michael Wagner	0.395	1.0	0.395	0.566
Bing Liu	0.825	1.0	0.825	0.904
Jim Smith	0.829	0.888	0.926	0.906
Lei Wang	0.863	0.920	0.932	0.926
Wei Wang	0.716	0.855	0.814	0.834
Bin Yu	0.658	1.0	0.658	0.794
average	0.810	0.966	0.836	0.883

Table 6.2: Accuracy for distinguishing references

We compare DISTINCT with DISTINCT without supervised learning, and DISTINCT using each of the two similarity measures: Set-resemblance [9] and random walk probabilities [64] (with and without supervised learning). Please notice that supervised learning is not used in [9] and [64]. For each approach except DISTINCT, we choose the min-sim that maximizes average accuracy. Figure 6.10 shows the average accuracy of each approach. DISTINCT improves the accuracy by about 20% compared with the approaches in [9] and [64]. The accuracy is improved by more than 10% with supervised learning, and 4% with combined similarity measure.

We visualize the results about "Wei Wang" in Figure 6.11. References corresponding to each author is shown in a gray box, together with his/her current affiliation and number of references. The arrows and small blocks indicate the mistakes made by DISTINCT. It can be seen that in general DISTINCT does a very good job in distinguishing references, although it makes some mistakes because of the linkages between references to different authors.

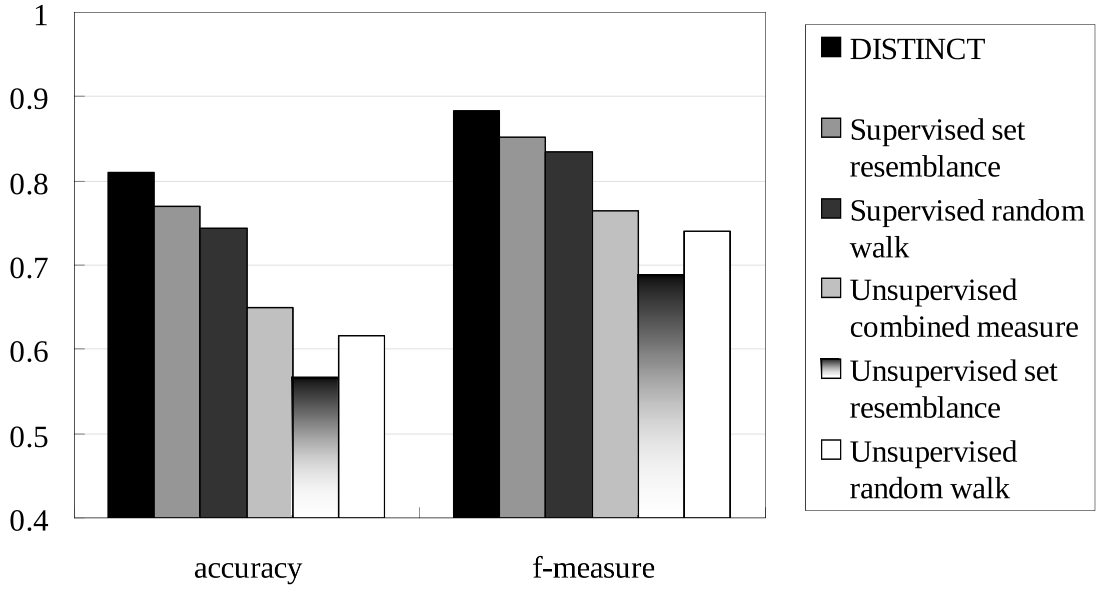


Figure 6.10: Accuracy and f-measure on real cases

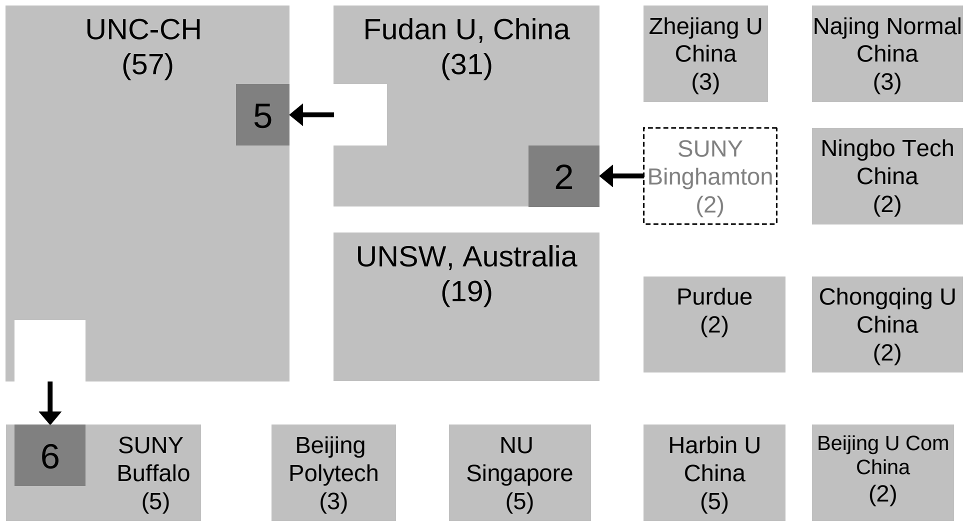


Figure 6.11: Groups of references of "Wei Wang"

6.5.2 Scalability

We test the scalability of DISTINCT w.r.t. the number of references to be reconciled. In the test on synthetic groups of references in DBLP database, there are 1141 groups of references to be reconciled. We apply DISTINCT on these 1141 problems of distinguishing references (min-sim=0.0005). Figure 6.12 is a scattered plot, where each point represents the running time and number of references of a group. One can see that although DISTINCT has $O(N^2 \log N)$ complexity as it uses agglomerative hierarchical clustering, it is highly efficient and only takes a few seconds to split 200 references. The average time for each group is 0.392 seconds.

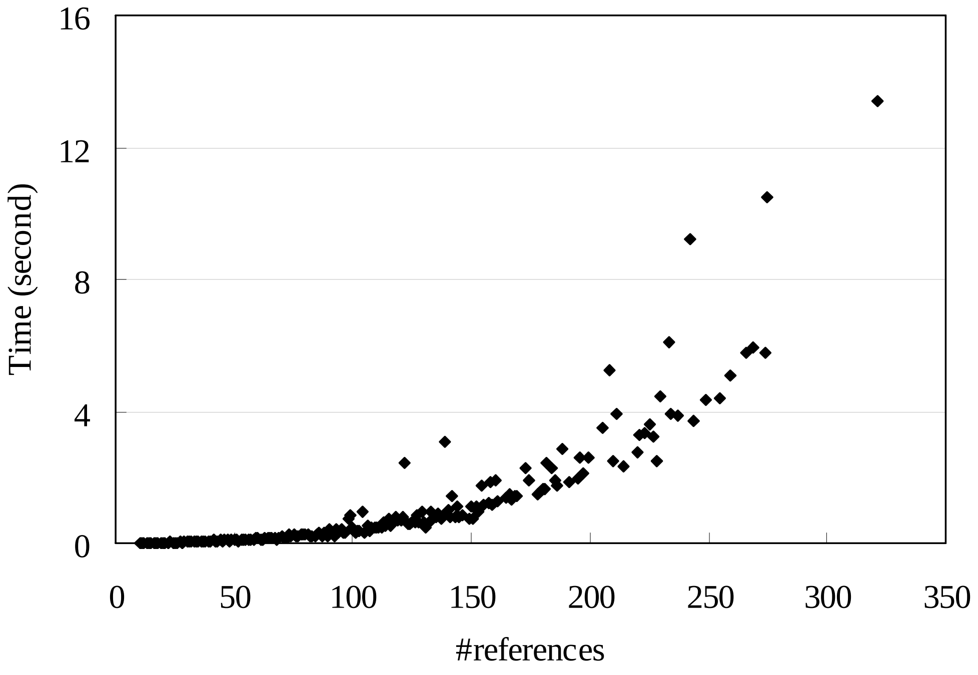


Figure 6.12: Scalability of DISTINCT on DBLP

6.6 Related Work

Reference splitting aims at finding different groups of references that refer to different objects among a set of identical references. This problem is most similar to record linkage [107], which aims at finding records with different contents but representing the same object or fact. Record linkage was first placed into a statistical framework in [42]. Most subsequent studies are based on computing pairwise similarities and collapsing similar records [4, 11, 21, 63]. There are substantial amount of work on using approximate string similarities or TF/IDF model for duplicate detection [21, 25, 52]. Some recent approaches [36, 99] consider the influences between different reconciliation decisions, and reconcile many candidate pairs simultaneously, although this could be very expensive for large-scale problems.

A record linkage approach usually follows a two-step procedure. It first identifies candidates of duplicate pairs of records, usually with a cheap measure (*e.g.*, textual similarity) such as the methods in [21, 52, 63].

Then it computes the similarity between each pair of candidate records based on their attribute values to decide whether they are duplicates.

However, the above procedure cannot be used to distinguish references with identical names. Because all references to be reconciled have identical names, textual similarity becomes useless, and there is no candidate. Because each reference is usually associated with very limited information, and even worse, two references to the same object may not have identical or similar attribute values since they appear in different contexts, one cannot judge their similarity by analyzing their attribute values.

There have been several studies on record linkage in relational databases [4, 9, 64]. Delphi is proposed in [4], which uses parents and children in a concept hierarchy help record linkage. Bhattacharya et al. propose RC-ER [9] which uses both the attributes and the neighbor tuples of records to match them. Kalashnikov et al. propose RelDC [64] which uses the random walk probabilities between different objects in a relational database to detects duplicates.

DISTINCT combines the two existing measures for similarities in record linkage in relational environments: (1) Set resemblance between the neighbor tuples [9], and (2) random walk probabilities [64]. Although DISTINCT uses existing measures, it works on a very different problem. The approaches in [9] and [64] aim at detecting duplicate objects with different names, and thus consider objects with identical names to be the same[4]. In comparison, DISTINCT aims at distinguishing references with identical names, and thus each reference being resolved must be considered as an individual object. This brings new challenges such as very limited information for each reference and no textual similarity. We use new techniques to address these challenges, which are shown to be highly effective by experiments.

Supervised learning has been used in record linkage in recent years [11]. Similar to statistics based approaches [42, 107], training data (pairs of duplicate or distinct records) is used to build a model for predicting whether a pair of records are duplicates. Among different machine learning approaches, SVM has been shown to be effective in [11].

6.7 Summary

In this chapter we study the problem of object distinction, *i.e.*, distinguishing references to objects with identical names. We develop a general methodology called DISTINCT for supervised composition of heterogeneous link analysis, that can fuse different types of linkages with differentiating weights, and use a combination of distinct similarity measures to assess the value of each linkage. DISTINCT combines two complementary

[4]In [9] authors with popular names (*e.g.*, "X. Wang") and authors who do not share coauthors are not considered as the same author.

measures for relational similarity (set resemblance of neighbor tuples and random walk probability). Using an SVM method to weigh different types of linkages, DISTINCT performs comprehensive linkage analysis based on linkages along different join paths. Our experiments show that DISTINCT can accurately distinguish different objects with identical names in real databases.

Chapter 7

Efficient Linkage-based Clustering

In Chapter 4 we introduce multi-relational clustering, which utilizes the multi-relational features and adopts the methodology of traditional clustering approaches. In Chapter 5 and Chapter 6 we explore the power of linkages in relational data mining. We believe linkages can also play an important role in clustering, because linkages contain rich semantic information and indicate the relationships among objects, which are essential for clustering objects.

In this chapter we study clustering based on linkages instead of object properties as in traditional clustering algorithms. The most crucial information for clustering is the similarities between objects. In our study the similarity between two objects is measured based on the similarities between the objects linked with them, *i.e.*, two objects are similar to each other if and only if they are linked to similar objects. We take advantage of the power law distribution of links, and develop a hierarchical structure called SimTree to represent similarities in multi-granularity manner. This method avoids the high cost of computing and storing pairwise similarities but still thoroughly explore relationships among objects. Experiments show the proposed approach achieves high efficiency, scalability, and accuracy in clustering multi-typed linked objects.

7.1 Introduction

As a process of partitioning data objects into groups according to their similarities with each other, clustering has been extensively studied for decades in different disciplines including statistics, pattern recognition, database, and data mining. There have been many clustering methods [1, 53, 75, 88, 98, 114], but most of them aim at grouping records in a *single table* into clusters using their own properties.

In many real applications, *linkages* among objects of different types can be the most explicit information available for clustering. For example, in a publication database (*i.e.*, PubDB) in Figure 7.1, one may want to cluster each type of objects (authors, institutions, publications, proceedings, and conferences/journals), in order to find authors working on different topics, or groups of similar publications, *etc.*. It is not so useful to cluster single type of objects (*e.g.*, authors) based only on the properties of them, as those properties often provide little information relevant to the clustering task. On the other hand, the linkages between different types of objects (*e.g.*, those between authors, papers and conferences) indicate the relationships between objects and can help cluster them effectively. Such *linkage-based clustering* is appealing in many applications. For example, an online movie store may want to cluster movies, actors, directors, reviewers, and renters, in order to improve its recommendation systems. In bioinformatics one may want to cluster genes, proteins, and their behaviors in order to discover their functions.

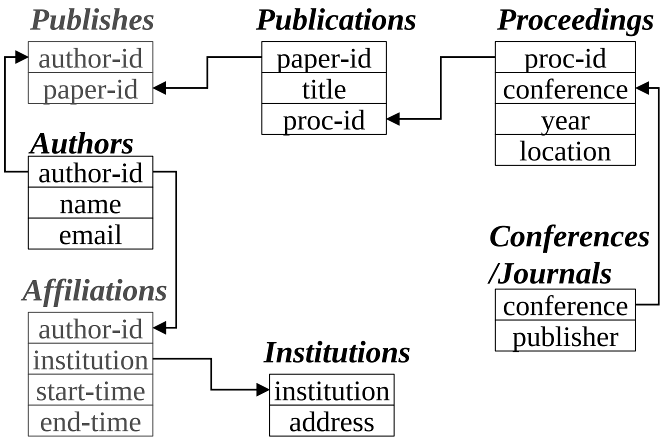


Figure 7.1: Schema of a publication database (PubDB)

Clustering based on multi-typed linked objects has been studied in multi-relational clustering [67, 112], in which the objects of each type are clustered based on the objects of other types linked with them. Consider the mini example in Figure 7.2. Authors can be clustered based on the conferences where they publish papers. However, such analysis is confined to direct links. For example, Tom publishes only SIGMOD papers, and John publishes only VLDB papers. Tom and John will have zero similarity based on direct links, although they may actually work on the same topic. Similarly, customers who have bought "*Matrix*" and those who have bought "*Matrix II*" may be considered dissimilar although they have similar interests.

The above example shows when clustering objects of one type, one needs to consider the similarities between objects of other types linked with them. For example, if it is known that SIGMOD and VLDB are similar, then SIGMOD authors and VLDB authors should be similar. Unfortunately, similarities between conferences may not be available, either. This problem can be solved by *SimRank* [62], in which the similarity between two objects is recursively defined as the average similarity between objects linked with them. For example, the similarity between two authors is the average similarity between the conferences in which they publish papers. In Figure 7.2 "sigmod" and "vldb" have high similarity because they share many coauthors, and thus Tom and John become similar because they publish papers in similar conferences. In contrast,

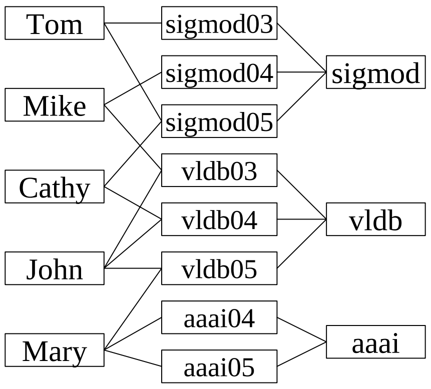


Figure 7.2: An example of linked objects in PubDB

John and Mary do not have high similarity even they are both linked with "vldb05".

Although SimRank provides a good definition for similarities based on linkages, it is prohibitively expensive in computation. In [62] an iterative approach is proposed to compute the similarity between every pair of objects, which has quadratic complexity in both time and space, and is impractical for large databases.

Is it necessary to compute and maintain pairwise similarities between objects? Our answer is no for the following two reasons. First, hierarchy structures naturally exist among objects of many types, such as the taxonomy of animals and hierarchical categories of merchandise. Consider the example of clustering authors according to their research. There are groups of authors working on the same research topic (*e.g.*, data integration or XML), who have high similarity with each other. Multiple such groups may form a larger group, such as the authors working on the same research area (*e.g.*, database vs. AI), who may have weaker similarity than the former. As a similar example, the density of linkages between clusters of articles and words is shown in Figure 7.3 (adapted from Figure 5 (b) in [20]). We highlight four dense regions with dashed boxes, and in each dense region there are multiple smaller and denser regions. The large dense regions correspond to high-level clusters, and the smaller denser regions correspond to low-level clusters within the high-level clusters.

Second, recent studies show that there exist power law distributions among the linkages in many domains, such as Internet topology and social networks [40]. Interestingly, based on our observation, such relationships also exist in the similarities between objects in interlinked environments. For example, Figure 7.4 shows the distribution of pairwise SimRank similarity values between 4170 authors in DBLP database (the plot shows portion of values in each 0.005 range of similarity value). It can be seen that majority of similarity entries have very small values which lie within a small range (0.005 – 0.015). While only a small portion of similarity entries have significant values, — 1.4% of similarity entries (about 123K of them) are greater than

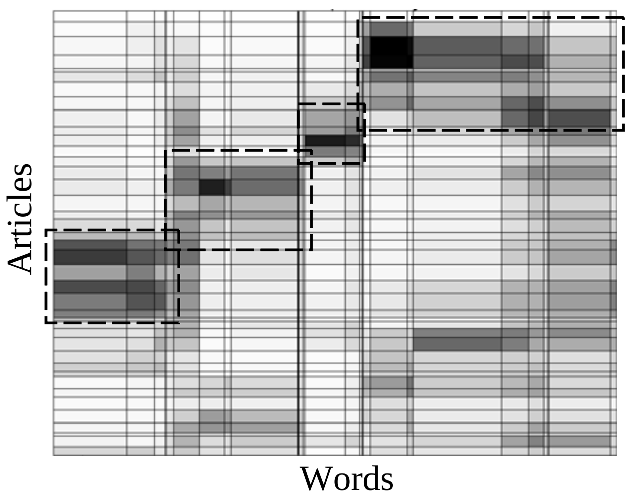


Figure 7.3: Density of linkages between articles and words

0.1, and these values will play the major role in clustering. Therefore, we want to design a data structure that stores the significant similarity values, and compresses those insignificant ones.

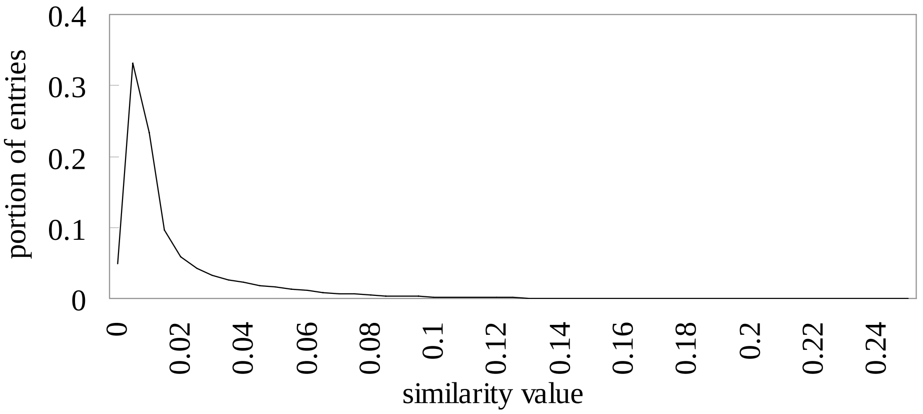


Figure 7.4: Portions of similarity values

Based on the above two observations, we propose a new hierarchical strategy to effectively prune the similarity space, which greatly speedups the identification of similar objects. Taking advantage of the power law distribution of linkages, we substantially reduce the number of pairwise similarities that need to be tracked, and the similarity between less similar objects will be approximated using aggregate measures.

We propose a hierarchical data structure called SimTree as a compact representation of similarities between objects. Each leaf node of a SimTree corresponds to an object, and each non-leaf node contains a group of lower-level nodes that are closely related to each other. SimTree stores similarities in a multi-granularity way by storing similarity between each two objects corresponding to sibling leaf nodes, and storing the overall similarity between each two sibling non-leaf nodes. Pairwise similarity is not pre-computed or maintained between objects that are not siblings. Their similarity, if needed, is derived based on the similarity information stored in the tree path. For example, consider the hierarchical categories of merchandise in Walmart. It is meaningful to compute the similarity between every two cameras, but not so meaningful to compute that for each camera and each TV, as an overall similarity between cameras and TVs should be sufficient.

Based on SimTree, we propose LinkClus, an efficient and accurate approach for linkage-based clustering. At the beginning LinkClus builds a SimTree for each type of objects in a bottom-up manner, by finding groups of objects (or groups of lower level nodes) that are similar to each other. Because inter-object similarity is not available yet, the similarity between two nodes are measured based on the intersection size of their neighbor objects. Thus the initial SimTrees cannot fully catch the relationships between objects (*e.g.*, some SIGMOD authors and VLDB authors have similarity 0).

LinkClus improves each SimTree with an iterative method, following the recursive rule that *two nodes are similar if they are linked with similar objects*. In each iteration it measures the similarity between two nodes in a SimTree by the average similarity between objects linked with them. For example, after one iteration SIGMOD and VLDB will become similar because they share many authors, which will then increase the similarities between SIGMOD authors and VLDB authors, and further increase that between SIGMOD and VLDB. We design an efficient algorithm for updating SimTrees, which merges the expensive similarity computations that go through the same paths in the SimTree. For a problem involving N objects and M linkages, LinkClus only takes $O(M(\log N)^2)$ time and $O(M+N)$ space (SimRank takes $O(M^2)$ time and $O(N^2)$ space).

Comprehensive experiments on both real and synthetic datasets are performed to test the accuracy and efficiency of LinkClus. It is shown that the accuracy of LinkClus is either very close or sometimes even better than that of SimRank, but with much higher efficiency and scalability. LinkClus also achieves much higher accuracy than other approaches on linkage-based clustering such as *ReCom* [105], and approach for approximating SimRank with high efficiency [43].

The rest of this chapter is organized as follows. We discuss related work in Section 7.2, and give an overview in Section 7.3. Section 7.4 introduces SimTree, the hierarchical structure for representing similarities. The algorithms for building SimTrees and computing similarities are described in Section 7.5. Our performance study is reported in Section 7.6, and this study is concluded in Section 7.7.

7.2 Related Work

Clustering has been extensively studied for decades in different disciplines including statistics, pattern recognition, database, and data mining, with many approaches proposed [1, 53, 75, 88, 98, 114]. Most existing clustering approaches aim at grouping objects in a single table into clusters, using properties of each object. Some recent approaches [67, 112] extend previous clustering approaches to relational databases and measures similarity between objects based on the objects joinable with them in multiple relations.

In many real applications of clustering, objects of different types are given, together with linkages among them. As the attributes of objects often provide very limited information, traditional clustering approaches can hardly be applied, and linkage-based clustering is needed, which is based on the principle that two objects are similar if they are linked with similar objects.

This problem is related to bi-clustering [23] (or co-clustering [31], cross-association [20]), which aims at finding dense submatrices in the relationship matrix of two types of objects. A dense submatrix corresponds to two groups of objects of different types that are highly related to each other, such as a cluster of genes and a cluster of conditions that are highly related. Unlike bi-clustering that involves no similarity computation, LinkClus computes similarities between objects based on their linked objects. Moreover, LinkClus works on a more general problem as it can be applied to a relational database with arbitrary schema, instead of two types of linked objects. LinkClus also avoids the expensive matrix operations often used in bi-clustering approaches.

A bi-clustering approach [31] is extended in [7], which performs agglomerative and conglomerative clustering simultaneously on different types of objects. However, it is very expensive, — quadratic complexity for two types and cubic complexity for more types.

Jeh and Widom propose SimRank [62], a linkage-based approach for computing the similarity between objects, which is able to find the underlying similarities between objects through iterative computations. Unfortunately SimRank is very expensive as it has quadratic complexity in both time and space. The authors also discuss a pruning technique for approximating SimRank, which only computes the similarity between a small number of preselected object pairs. In the extended version of [62] the following heuristic is used: Only similarities between pairs of objects that are linked with same objects are computed. With this heuristic, in Figure 7.2 the similarity between SIGMOD and VLDB will never be computed. Neither will the similarity between Tom and John, Tom and Mike, *etc.*. In general, it is very challenging to identify the right pairs of objects at the beginning, because many pairs of similar objects can only be identified after computing similarities between other objects. In fact this is the major reason that we adopt the recursive definition of similarity and use iterative methods.

A method is proposed in [43] to perform similarity searches by approximating SimRank similarities. It creates a large sample of random walk paths from each object and uses them to estimate the SimRank similarity between two objects when needed. It is suitable for answering similarity queries. However, very large samples of paths are needed for making accurate estimations for similarities. Thus it is very expensive in both time and space to use this approach for clustering a large number of objects, which requires computing similarities between numerous pairs of objects.

Wang et al. propose ReCom [105], an approach for clustering inter-linked objects of different types. ReCom first generates clusters using attributes and linked objects of each object, and then repeatedly refines the clusters using the clusters linked with each object. Compared with SimRank that explores pairwise similarities between objects, ReCom only explores the neighbor clusters and does not compute similarities between objects. Thus it is much more efficient but much less accurate than SimRank.

LinkClus is also related to hierarchical clustering [53, 98]. However, they are fundamentally different. Hierarchical clustering approaches use some similarity measures to put objects into hierarchies. While LinkClus uses hierarchical structures to represent similarities.

7.3 Overview

Linkage-based clustering is based on the principle that two objects are similar if they are linked with similar objects. For example, in a publication database (Figure 7.2), two authors are similar if they publish similar papers. The final goal of linkage-based clustering is to divide objects into clusters using such similarities. Figure 7.5 shows an example of three types of linked objects, and clusters of similar objects which are inferred from the linkages. It is important to note that objects 12 and 18 do not share common neighbors, but they are linked to objects 22 and 24, which are similar because their common linkages to 35, 37 and 38.

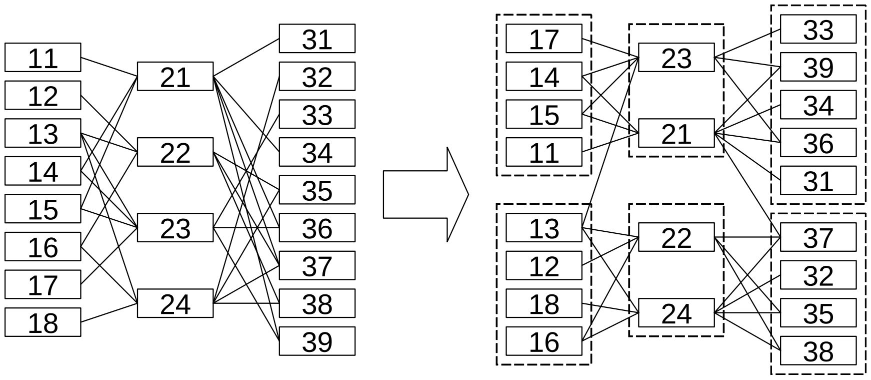


Figure 7.5: Finding groups of similar objects

In order to capture the inter-object relationships as in the above example, we adopt the recursive definition of similarity in SimRank [62], in which the similarity between two objects x and y is defined as the average similarity between the objects linked with x and those linked with y.

As mentioned in the introduction, a hierarchical structure can capture the hierarchical relationships among objects, and can compress the majority of similarity values which are insignificant. Thus we use SimTree, a hierarchical structure for storing similarities in a multi-granularity way. It stores detailed similarities between closely related objects, and overall similarities between object groups. We generalize the similarity measure in [62] to hierarchical environments, and propose an efficient and scalable algorithm for computing similarities based on the hierarchical structure. Each node in a SimTree has at most c children, where c is a constant and is usually between 10 and 20. Given a database containing two types of objects, N objects of each type, and M linkages between them, our algorithm takes $O(Nc + M)$ space and $O(M \cdot (\log_c N)^2 \cdot c^2)$ time. This is affordable for very large databases.

7.4 SimTree: Hierarchical Representation of Similarities

In this section we describe SimTree, a new hierarchical structure for representing similarities between objects. Each leaf node of a SimTree represents an object (by storing its ID), and each non-leaf node has a set of child nodes, which are a group of closely related nodes of one level lower. An example SimTree is shown in Figure 7.6. The small gray circles represent leaf nodes, which must appear at the same level (which is level-0, the bottom level). The dashed circles represent non-leaf nodes. Each non-leaf node has at most c child nodes, where c is a small constant. Between each pair of sibling nodes n_i and n_j there is an undirected edge (n_i, n_j). (n_i, n_j) is associated with a real value $s(n_i, n_j)$, which is the average similarity between all objects linked with n_i (or with its descendant objects if n_i is a non-leaf node) and those with n_j. $s(n_i, n_j)$ represents the overall similarity between the two groups of objects contained in n_i and n_j.

Another view of the same SimTree is shown in Figure 7.7, which better visualizes the hierarchical structure. The similarity between each pair of sibling leaf nodes is stored in the SimTree. While the similarity between two non-sibling leaf nodes is estimated using the similarity between their ancestor nodes. For example, suppose the similarity between n_7 and n_8 is needed, which is the average similarity between objects linked with n_7 and those with n_8. One can see that n_4 (or n_5) contains a small group of leaf nodes including n_7 (or n_8), and we have computed $s(n_4, n_5)$ which is the average similarity between objects linked with these two groups of leaf nodes. Thus LinkClus uses $s(n_4, n_5)$ as the estimated similarity between n_7 and n_8. In a real application such as clustering products in Walmart, n_7 may correspond to a camera and n_8 to

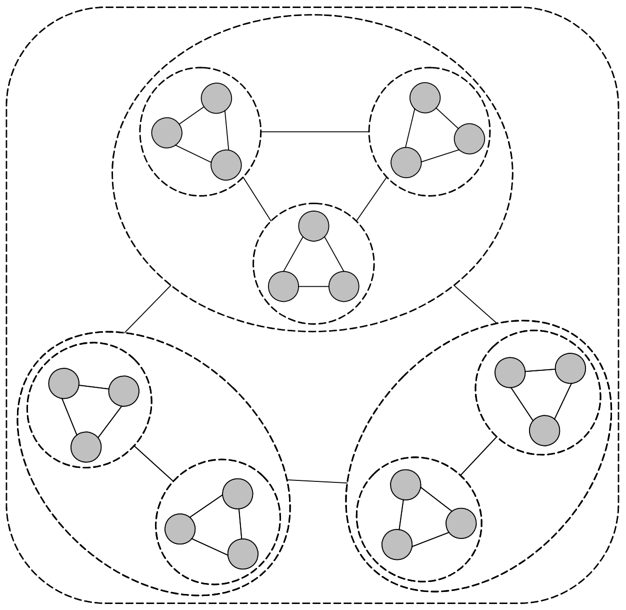


Figure 7.6: Structure of an example SimTree

a TV. We can estimate their similarity using the overall similarity between cameras and TVs, which may correspond to n_4 and n_5, respectively. Similarly when the similarity between n_7 and n_9 is needed, LinkClus uses $s(n_1, n_2)$ as an estimation.

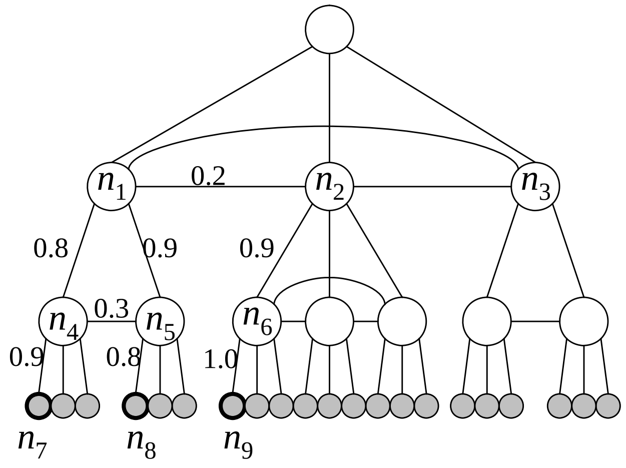


Figure 7.7: Another view of the SimTree

Such estimation is not always accurate, because a node may have different similarities to other nodes compared with its parent. LinkClus makes some adjustments to compensate for such differences, by associating a value to the edge between each node and its parent. For example, the edge (n_7, n_4) is associated with a real value $s(n_7, n_4)$, which is the ratio between *(1)* the average similarity between n_7 and all leaf nodes except n_4's descendants, and *(2)* the average similarity between n_4 and those nodes. Similarly we can define $s(n_4, n_1)$, $s(n_6, n_2)$, *etc.*. When estimating the similarity between n_7 and n_9, we use $s(n_1, n_2)$ as a basic estimation, use $s(n_4, n_1)$ to compensate for the difference between similarities involving n_4 and those involving n_1, and use $s(n_7, n_4)$ to compensate for n_7. The final estimation is $s(n_7, n_4) \cdot s(n_4, n_1) \cdot s(n_1, n_2) \cdot s(n_6, n_2) \cdot s(n_9, n_6) = 0.9 \cdot 0.8 \cdot 0.2 \cdot 0.9 \cdot 1.0 = 0.1296$.

In general, the similarity between two leaf nodes w.r.t. a SimTree is the product of the values of all edges on the path between them. Because this similarity is defined based on the path between two nodes, we call it *path-based similarity.*

Definition 16 ***(Path-based Node Similarity)*** *Suppose two leaf nodes n_1 and n_k in a* SimTree *are connected by path $n_1 \rightarrow \ldots \rightarrow n_i \rightarrow n_{i+1} \rightarrow \ldots \rightarrow n_k$, in which n_i and n_{i+1} are siblings and all other edges are between nodes and their parents. The path-based similarity between n_1 and n_k is*

$$sim_p(n_1, n_k) = \prod_{j=1}^{k-1} s(n_j, n_{j+1}) \tag{7.1}$$

Each node has similarity 1 with itself ($sim_p(n, n) = 1$).

Please note that within a path in Definition 16, there is only one edge that is between two sibling nodes, whose similarity is used as the basic estimation. The other edges are between parent and child nodes whose similarities are used for adjustments.

7.5 Building SimTrees

The input to LinkClus are objects of different types, with linkages between them. LinkClus maintains a SimTree for each type of objects to represent similarities between them. Each object is used as a leaf node in a SimTree. Figure 7.8 shows the leaf nodes created from objects of two types, and the linkages between them.

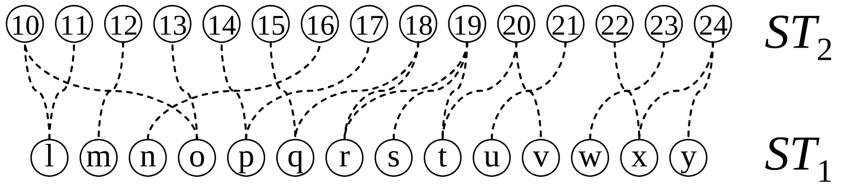


Figure 7.8: Leaf nodes in two SimTrees

Initially each object has similarity 1 to itself and 0 to others. LinkClus first builds SimTrees using the initial similarities. These SimTrees may not fully catch the real similarities between objects, because inter-object similarities are not considered. LinkClus uses an iterative method to improve the SimTrees, following the principle that two objects are similar if and only if they are linked with similar objects. It repeatedly updates each SimTree using the following rule: The similarity between two nodes n_i and n_j is the average similarity between objects linked with n_i and those linked with n_j. The structure of each SimTree is also adjusted during each iteration by moving similar nodes together. In this way the similarities are refined in each iteration, and the relationships between objects can be discovered gradually.

7.5.1 Initializing SimTrees Using Frequent Pattern Mining

The first step of LinkClus is to initialize SimTrees using the linkages as shown in Figure 7.8. Although no inter-object similarities are available at this time, the initial SimTrees should still be able to group related objects or nodes together, in order to provide a good base for further improvements.

Because only leaf nodes are available at the beginning, we initialize SimTrees from bottom level to top level. At each level, we need to efficiently find groups of tightly related nodes, and use each group as a node of the upper level. Consider a group of nodes $g = \{n_1, \ldots, n_k\}$. Let $neighbor(n_i)$ denote the set of objects linked with node n_i. Initially there are no inter-object similarities, and whether two nodes are similar depends on whether they are co-linked with many objects. Therefore, we define the tightness of group g as the number of objects that are linked with all group members, *i.e.*, the size of intersection of $neighbor(n_1), \ldots, neighbor(n_k)$.

The problem of finding groups of nodes with high tightness can be reduced to the problem of finding frequent patterns [3]. A tight group is a set of nodes that are co-linked with many objects of other types, just like a frequent pattern is a set of items that co-appear in many transactions. Figure 7.9 shows an example which contains four nodes n_1, n_2, n_3, n_4 and objects linked with them. The nodes linked with each object are converted into a transaction, which is shown on the right side. The following lemma shows that, the number of objects that are linked with all members of a group g is equal to the support of the pattern corresponding to g in the transactions.

Lemma 6 *Given a set of nodes $\{n_1, \ldots, n_N\}$ and their neighbor objects, suppose we are looking for groups of nodes with high tightness among them. If we convert the nodes linked with each neighbor object into a transaction, then a group of nodes has tightness T if and only if the corresponding pattern has support T in the transactions.*

PROOF. *(Please refer to Figure 7.9.) For a group of nodes $g = \{n'_1, \ldots, n'_k\}$, the tightness of this group*

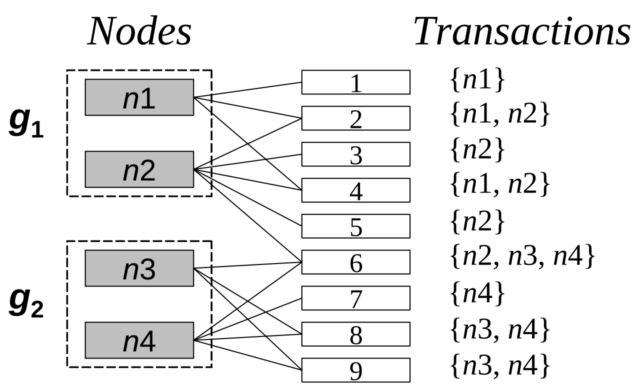


Figure 7.9: Groups of tightly related nodes

T is the number of objects that are linked to all nodes in g. If an object o is linked to all nodes in g, then the pattern $(n'_1, \ldots, n'_k)$ appears in the transaction of o, and this pattern appears in the transaction of o only if all nodes in g are linked to o. Therefore, the tightness of a group is equal to the support of the corresponding pattern. ■

For example, nodes n_1 and n_2 are co-linked with two objects (#2 and #4), and pattern $\{n_1, n_2\}$ has support 2 (*i.e.*, appear twice) in the transactions.

Let *support*(g) represent the number of objects linked with all nodes in g. When building a SimTree, we want to find groups with high support and at least min_size nodes. For two groups g and g' such that $g \subset g'$ and *support*(g) = *support*(g'), we prefer g'. Frequent pattern mining has been studied for a decade with many efficient algorithms. We can either find groups of nodes with support greater than a threshold using a frequent closed pattern mining approach [104], or find groups with highest support using a top-k frequent closed pattern mining approach [58]. LinkClus uses the approach in [104] which is very efficient on large datasets.

Now we describe the procedure of initializing a SimTree. Suppose we have built N_l nodes at level-l of the SimTree, and want to build the nodes of level-$(l+1)$. Because each node can have at most c child nodes, and because we want to leave some space for further adjustment of the tree structure, we control the number of level-$(l+1)$ nodes to be between $\frac{N_l}{c}$ and $\frac{\alpha N_l}{c}$ $(1 < \alpha \leq 2)$. We first find groups of level-l nodes with sufficiently high support. Since there are usually many such groups, we select $\frac{\alpha N_l}{c}$ non-overlapping groups with high support in a greedy way, by repeatedly selecting the group with highest support that is not overlapped with previously selected groups. After selecting $\frac{\alpha N_l}{c}$ groups, we create a level-$(l+1)$ node based on each group. However, these groups usually cover only part of all level-l nodes. For each level-l node n_i that does not belong to any group, we want to put n_i into the group that is most connected with

n_i. For each group g, we compute the number of objects that are linked with both n_i and some members of g, which is used to measure the connection between n_i and g. We assign n_i to the group with highest connection to n_i.

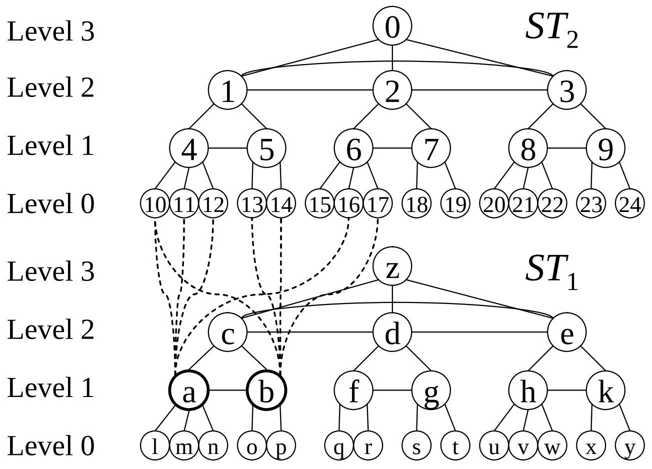


Figure 7.10: Some linkages between two SimTrees

Figure 7.10 shows the two SimTrees built upon the leaf nodes in Figure 7.8. The dashed lines indicate the leaf nodes in ST_2 that are linked with descendants of two non-leaf nodes n_a and n_b in ST_1. After building the initial SimTrees, LinkClus computes the similarity value associated with each edge in the SimTrees. As defined in Section 7.4, the similarity value of edge (n_a, n_b), $s(n_a, n_b)$, is the average similarity between objects linked with descendants of n_a and those of n_b. Because initially the similarity between any two different objects is 0, $s(n_a, n_b)$ can be easily computed based on the number of objects that are linked with both the descendants of n_a and those of n_b, without considering pairwise similarities. Similarly, the values associated with edges between child and parent nodes can also be computed easily.

7.5.2 Refining Similarity between Nodes

The initial SimTrees cannot fully catch the real similarities, because similarities between objects are not considered when building them. Therefore, LinkClus repeatedly updates the SimTrees, following the principle that the similarity between two nodes in a SimTree is the average similarity between the objects linked with them, which is indicated by other SimTrees. This is formally defined in this subsection.

We use $[n \sim n']$ to denote the linkage between two nodes n and n' in different SimTrees. We say there is a linkage between a non-leaf node n in ST_1 and a node n' in ST_2, if there are linkages between the descendant leaf nodes of n and the node n'. Figure 7.10 shows the linkages between n_a, n_b and leaf nodes in ST_2. In order to track the number of original linkages involved in similarity computation, we assign a weight to each linkage. By default the weight of each original linkage between two leaf nodes is 1. The weight of linkage $[n \sim n']$ is the total number of linkages between the descendant leaf nodes of n and n'.

In each iteration LinkClus updates the similarity between each pair of sibling nodes (e.g., n_a and n_b) in each SimTree, using the similarities between the objects linked with them in other SimTrees. The similarity between n_a and n_b is the average path-based similarity between the leaf nodes linked with n_a ($\{n_{10}$,n_{11},n_{12},$n_{16}\}$) and those with n_b ($\{n_{10}$,n_{13},n_{14}, $n_{17}\}$). Because this similarity is based on linked objects, we call it *linkage-based similarity.* n_a (or n_b) may have multiple linkages to a leaf node n_i in ST_2, if more than one descendants of n_a are linked with n_i. Thus the leaf nodes in ST_2 linked with n_a are actually a multi-set, and the frequency of each leaf node n_i is $weight([n_a \sim n_i])$, which is the number of original linkages between n_a and n_i. The linkage-based similarity between n_a and n_b is defined as the average path-based similarity between these two multi-sets of leaf nodes, and in this way each original linkage plays an equal role.

Definition 17 ***(Linkage-based Node Similarity)*** *Suppose a* SimTree *ST is linked with* SimTrees *$ST_1, \ldots, ST_K$. For a node n in ST, let $NB_{ST_k}(n)$ denote the multi-set of leaf nodes in ST_k linked with n. Let $w_{n'n''}$ represent $weight([n' \sim n''])$. For two nodes n_a and n_b in ST, their linkage-based similarity $sim_l(n_a, n_b)$ is the average similarity between the multi-set of leaf nodes linked with n_a and that of n_b. We decompose the definition into several parts for clarity.*

(The total weights between $NB_{ST_k}(n_a)$ and $NB_{ST_k}(n_b)$)

$$weight_{ST_k}(n_a, n_b) = \sum_{n \in NB_{ST_k}(n_a)} \sum_{n' \in NB_{ST_k}(n_b)} w_{n_a n} \cdot w_{n_b n'}$$

(The sum of weighted similarity between them)

$$sum_{ST_k}(n_a, n_b) = \sum_{n \in NB_{ST_k}(n_a)} \sum_{n' \in NB_{ST_k}(n_b)} w_{n_a n} \cdot w_{n_b n'} \cdot sim_p(n, n')$$

(The linkage-based similarity between n_a and n_b w.r.t. ST_k)

$$sim_{ST_k}(n_a, n_b) = \frac{sum_{ST_k}(n_a, n_b)}{weight_{ST_k}(n_a, n_b)}$$

(The final definition of $sim_l(n_a, n_b)$)

$$sim_l(n_a, n_b) = \frac{1}{K} \sum_{k=1}^{K} sim_{ST_k}(n_a, n_b). \tag{7.2}$$

Equation (7.2) shows that if a SimTree ST is linked with multiple SimTrees, each linked SimTree plays an equal role in determining the similarity between nodes in ST. The user can also use different weights for different SimTrees according to the semantics.

7.5.3 Aggregation-based Similarity Computation

The core part of LinkClus is how to iteratively update each SimTree, by computing linkage-based similarities between different nodes. This is also the most computation-intensive part in linkage-based clustering. Definition 17 provides a brute-force method to compute linkage-based similarities. However, it is very expensive. Suppose each of two nodes n_a and n_b is linked with m leaf nodes. It takes $O(m^2 \log_c N)$ to compute $sim_l(n_a, n_b)$ ($\log_c N$ is the height of SimTree). Because some high-level nodes are linked with $\Theta(N)$ objects, this brute-force method requires $O(N^2 \log_c N)$ time, which is unaffordable for large databases.

Fortunately, we find that the computation of different path-based similarities can be merged if these paths are overlapped with each other.

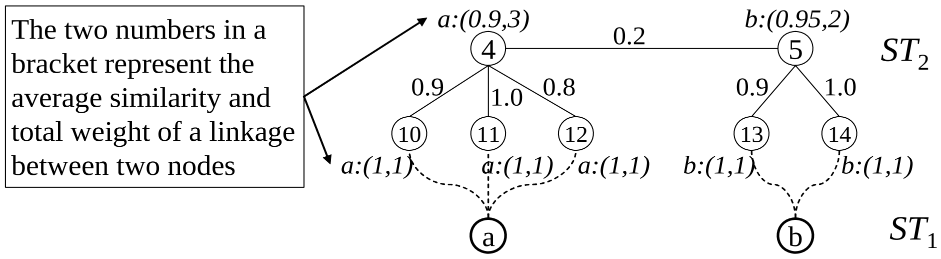

The two numbers in a bracket represent the average similarity and total weight of a linkage between two nodes
a:(0.9,3)
b:(0.95,2)
0.2
4
5
ST_2
0.9
1.0
0.8
0.9
1.0
10
11
12
13
14
a:(1,1)
a:(1,1)
a:(1,1)
b:(1,1)
b:(1,1)
a
b
ST_1

Figure 7.11: Computing similarity between nodes

Example 2 *A simplified version of Figure 7.10 is shown in Figure 7.11, where $sim_l(n_a, n_b)$ is the average path-based similarity between each node in $\{n_{10}, n_{11}, n_{12}\}$ and each in $\{n_{13}, n_{14}\}$. For each node $n_k \in \{n_{10}, n_{11}, n_{12}\}$ and $n_l \in \{n_{13}, n_{14}\}$, their path-based similarity $sim_p(n_k, n_l) = s(n_k, n_4) \cdot s(n_4, n_5) \cdot s(n_5, n_l)$. All these six path-based similarities involve $s(n_4, n_5)$. Thus $sim_l(n_a, n_b)$, which is the average of them, can be written as*

$$sim_l(n_a, n_b) = \frac{\sum_{k=10}^{12} s(n_k, n_4)}{3} \cdot s(n_4, n_5) \cdot \frac{\sum_{l=13}^{14} s(n_l, n_5)}{2}. \tag{7.3}$$

Equation (7.3) contains three parts: the average similarity between n_a and descendants of n_4, $s(n_4, n_5)$, and *the average similarity between n_b and descendants of n_5. Therefore, we pre-compute the average similarity and total weights between n_a,n_b and n_4,n_5, as shown in Figure 7.11. (The original linkages between leaf nodes do not affect similarities and thus have similarity 1.) We can compute $sim_l(n_a, n_b)$ using such aggregates, i.e., $sim_l(n_a, n_b) = \frac{0.9+1.0+0.8}{3} \times 0.2 \times \frac{0.9+1.0}{2} = 0.9 \times 0.2 \times 0.95 = 0.171$, and this is the average similarity between $3 \times 2 = 6$ pairs of leaf nodes. This is exactly the same as applying Definition 2 directly. But now we have avoided the pairwise similarity computation, since only the edges between siblings and parent-child are involved.*

This mini example shows the basic idea of computing linkage-based similarities. In a real problem n_a and n_b are often linked with many leaf nodes lying in many different branches of the SimTrees, which makes the computation much more complicated. The basic idea is still to merge computations that share common paths in SimTrees.

To facilitate our discussion, we introduce a simple data type called simweight, which is used to represent the similarity and weight associated with a linkage. A simweight is a pair of real numbers $\langle s, w \rangle$, in which s is the similarity of a linkage and w is its weight. We define two operations of simweight that are very useful in LinkClus.

Definition 18 ***(Operations of* simweight*)***
The operation of addition is used to combine two simweights corresponding to two linkages. The new similarity is the weighted average of their similarities, and the new weight is the sum of their weights:

$$\langle s_1, w_1 \rangle + \langle s_2, w_2 \rangle = \langle \frac{s_1 \cdot w_1 + s_2 \cdot w_2}{w_1 + w_2}, w_1 + w_2 \rangle. \tag{7.4}$$

The operation of multiplication is used to compute the weighted average similarity between two sets of leaf nodes. The new weight $w_1 \cdot w_2$ represents the number of pairs of leaf nodes between the two sets.

$$\langle s_1, w_1 \rangle \times \langle s_2, w_2 \rangle = \langle s_1 \cdot s_2, w_1 \cdot w_2 \rangle. \tag{7.5}$$

Lemma 7 *The laws of commutation, association, and distribution hold for the operations of simweight.*

Proof. *Can be derived from definitions.* ■

LinkClus uses a simweight to represent the relationship between two nodes in different SimTrees. We use $NB_{ST}(n)$ to denote the multi-set of leaf nodes in ST linked with node n. For example, in Figure 7.10 $NB_{ST_2}(n_a)$ =\{n_{10},n_{11},n_{12},n_{16}\} and $NB_{ST_2}(n_b)$ = \{n_{10},n_{13},n_{14},n_{17}\}.

We first define the weight and similarity of a linkage between two non-leaf nodes in two SimTrees. A non-leaf node represents the set of its child nodes. Therefore, for a node n_a in ST_1 and a non-leaf node n_i in ST_2, the weight and similarity of linkage $[n_a \sim n_i]$ is the sum of weights and weighted average similarity between their child nodes. Furthermore, according to Definition 16, the similarity between two non-sibling nodes n_i and n_j on the same level of ST_2 can be calculated as

$$sim_p(n_i, n_j) = s(n_i, parent(n_i)) \cdot sim_p(parent(n_i), parent(n_j)) \cdot s(n_j, parent(n_j)).$$

Thus we also incorporate $s(n_i, parent(n_i))$ (*i.e.*, the ratio between average similarity involving n_i and that involving $parent(n_i)$) into the definition of linkage $[n_a \sim n_i]$. We use $sw_{n_a n_i}$ to denote the simweight of $[n_a \sim n_i]$.

Definition 19 *Let n_a be a node in* SimTree *ST_1 and n_i be a non-leaf node in ST_2. Let $children(n_i)$ be all child nodes of n_i. The* simweight *of linkage $[n_a \sim n_i]$ is defined as*

$$sw_{n_a n_i} = \sum_{\hat{n} \in children(n_i)} \langle s(\hat{n}, n_i), 1 \rangle \times sw_{n_a \hat{n}}. \tag{7.6}$$

(In Equation (7.6) we use $\langle x, 1 \rangle \times \langle s, w \rangle$ as a convenient notation for $\langle x \cdot s, w \rangle$. Figure 7.11 and Figure 7.12 show $sw_{n_a n_i}$ and $sw_{n_b n_i}$ for each node n_i in ST_2.)

Using Definition 19, we formalize the idea in Example 2 as follows.

Lemma 8 *For two nodes n_a and n_b in* SimTree *ST_1, and two sibling non-leaf nodes n_i and n_j in ST_2, the average similarity and total weight between the descendant objects of n_i linked with n_a and those of n_j linked with n_b is*

$$sw_{n_a n_i} \times \langle s(n_i, n_j), 1 \rangle \times sw_{n_b n_j}.$$

(This corresponds to Equation (7.3) if $i = 4$ and $j = 5$.)

Proof. *Can be derived from definitions.* ■

We outline the procedure for computing the linkage-based similarity between n_a and n_b (see Figure 7.12). $sim_l(n_a, n_b)$ is the average similarity between $NB_{ST_2}(n_a)$ and $NB_{ST_2}(n_b)$. We first compute the aggregated simweights $sw_{n_a n}$ and $sw_{n_b n}$ for each node n in ST_2, if n is an ancestor of any node in $NB_{ST_2}(n_a)$ or $NB_{ST_2}(n_b)$, as shown in Figure 7.12. Consider each pair of sibling nodes n_i and n_j in ST_2 (e.g., n_4 and n_5), so that n_i is linked with n_a and n_j with n_b. According to Lemma 2, the average similarity

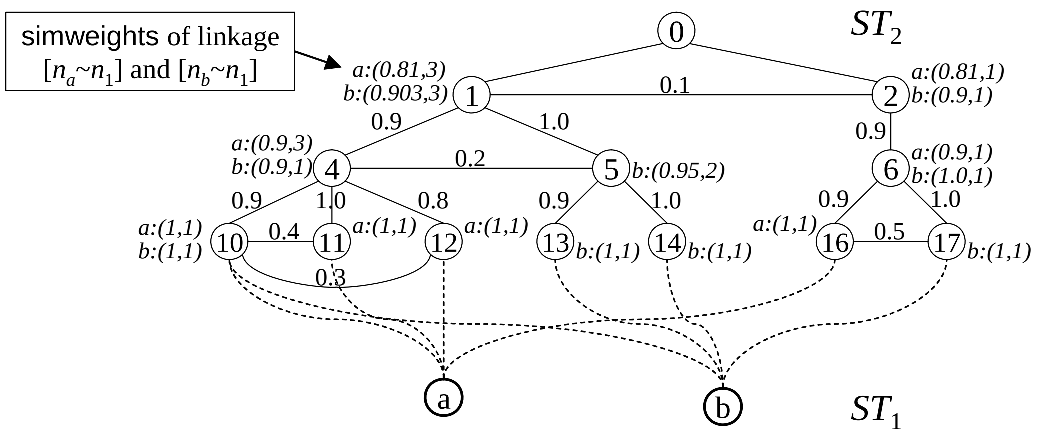


Figure 7.12: Computing similarity between nodes

and total weight between the descendant objects of n_i linked with n_a and those of n_j linked with n_b is $sw_{n_a n_i} \times \langle s(n_i, n_j), 1\rangle \times sw_{n_b n_j}$. For example, $sw_{n_a n_4} = \langle 0.9, 3\rangle$ (where the weight is 3 as n_a can reach n_4 via n_{10}, n_{11} or n_{12}), $sw_{n_b n_5} = \langle 0.95, 2\rangle$, and $s(n_4, n_5) = 0.2$. Thus $sw_{n_a n_4} \times \langle s(n_4, n_5), 1\rangle \times sw_{n_b n_5} = \langle 0.171, 6\rangle$ (as in Example 2), which represents the average similarity and total weights between $\{n_{10},n_{11},n_{12}\}$ and $\{n_{13},n_{14}\}$. We note that the weight is 6 as there are 6 paths between leaf nodes under n_4 linked with n_a and those under n_5 linked with n_b.

From the above example it can be seen that the effect of similarity between every pair leaf nodes in ST_2 will be captured when evaluating their ancestors that are siblings. For any two leaf nodes $\hat{n}_i$ and $\hat{n}_j$ in ST_2, there is only one ancestor of $\hat{n}_i$ and one of $\hat{n}_j$ that are siblings. Thus every pair of $\hat{n}_i$, $\hat{n}_j$ ($\hat{n}_i \in NB_{ST_2}(n_a)$, $\hat{n}_j \in NB_{ST_2}(n_b)$) is counted exactly once, and no redundant computation is performed. In general, $sim_l(n_a, n_b)$ can be computed using Theorem 2.

Theorem 2 ***(Sibling-pair based Similarity Computation)*** *Suppose n_a and n_b are two nodes in* SimTree *ST_1. Let $NB_{ST_2}(n_a)$ and $NB_{ST_2}(n_b)$ be the multi-sets of leaf nodes in* SimTree *ST_2 linked with n_a and n_b, respectively.*

$$\langle sim_l(n_a, n_b),\ \text{weight is ignored}\rangle =$$
$$\sum_{n \in ST_2} \sum_{n_i, n_j \in children(n), n_i \neq n_j} sw_{n_a n_i} \times \langle s(n_i, n_j), 1\rangle \times sw_{n_b n_j} + \sum_{n_i \in NB_{ST_2}(n_a) \bigcap NB_{ST_2}(n_b)} sw_{n_a n_i} \times sw_{n_b n_i}. \quad (7.7)$$

Proof. *Suppose $\langle s_{ab}, w_{ab}\rangle$ is the weighted average similarity and sum of products of weights between $NB_{ST_2}(n_a)$ and $NB_{ST_2}(n_b)$. We need to prove that $\langle s_{ab}, w_{ab}\rangle = \langle \check{s}, \check{w}\rangle$.*

Let us first consider that part in $\langle s_{ab}, w_{ab}\rangle$ that comes from the linkages from n_a and n_b to the same node *in ST_2. For each node $\hat{n}$ in $NB_{ST_2}(n_a) \bigcap NB_{ST_2}(n_b)$, the linkage $[n_a \sim \hat{n}]$ (or $[n_b \sim \hat{n}]$) has similarity 1. Therefore, the part in $\langle s_{ab}, w_{ab} \rangle$ that comes from linkages from n_a and n_b to $NB_{ST_2}(n_a) \bigcap NB_{ST_2}(n_b)$ is $\sum_{n_i \in NB_{ST_2}(n_a) \bigcap NB_{ST_2}(n_b)} sw_{n_a n_i} \times sw_{n_b n_i}$.*

Then let us consider two different leaf nodes $n_i \in NB_{ST_2}(n_a)$ and $n_j \in NB_{ST_2}(n_b)$. Let $\check{n}_{ij}$ be the least common ancestor of n_i and n_j. In equation (7.7) we consider the product of sim-weight *of the two linkages $[n_a \sim n_i]$ and $[n_b \sim n_j]$ exactly once. That is when considering child nodes of $\check{n}_{ij}$.*

Then we will show that the product of sim-weight *of the two linkages $[n_a \sim n_i]$ and $[n_b \sim n_j]$ is correctly added into $\langle s_{ab}, w_{ab} \rangle$. Let $\check{n}_i$ and $\check{n}_j$ be ancestors of n_i and n_j which are children of $\check{n}_{ij}$. From Definition 16 we know that $sim(n_i, n_j)$ is the product of similarities on all edges on the path $p(n_i, n_j)$ from n_i to n_j. p_{ij} can be decomposed into three parts: $p(n_i, \check{n}_i)$, $\check{n}_i \to \check{n}_j$, and $p(\check{n}_j, n_j)$. From Definition 19, we can see that $simweight([n_a \sim n_i])$ takes care of the product of similarities on $p(n_i, \check{n}_i)$. So does $simweight([n_b \sim n_j])$ for $p(\check{n}_j, n_j)$. And $s(n_i, n_j)$ is the product of similarity of path $\check{n}_i \to \check{n}_j$. Therefore, the similarity between n_i and n_j within ST_2 and the product of weights of $[n_a \sim n_i]$ and $[n_b \sim n_j]$ are correctly added into $\langle s_{ab}, w_{ab} \rangle$ using equation (7.7).*

We have shown that each pair of leaf nodes $n_i \in NB_{ST_2}(n_a)$ and $n_j \in NB_{ST_2}(n_b)$ is considered exactly once, and the similarity between them and the product of their weights are correctly added. Therefore, $\langle \check{s}, \check{w} \rangle$ represents the weighted average similarity and the total product of weights between $NB_{ST_2}(n_a)$ and $NB_{ST_2}(n_b)$.

Please note that the this theorem does not depend on any particular definitions of similarity associated with edges. We can define any similarity for edges. We can also define additive edge weights instead of similarities. For example, we can use logarithm of similarity as the weight for an edge. ■

The first term of equation (7.7) corresponds to similarities between different leaf nodes. For all leaf nodes under n_i linked with n_a and those under n_j linked with n_b, the effect of pairwise similarities between them is aggregated together as computed in the first term. The second term of equation (7.7) corresponds to the leaf nodes linked with both n_a and n_b. Only similarities between sibling nodes are used in equation (7.7), and thus we avoid the tedious pairwise similarity computation in Definition 17. In order to compute the linkage-based similarities between nodes in ST_1, it is sufficient to compute aggregated similarities and weights between nodes in ST_1 and nodes in other SimTrees. This is highly efficient in time and space as shown in Section 7.5.5.

Now we describe the procedure of computing $sim_l(n_a, n_b)$ based on Theorem 2.

Step 1: Attach the simweight of each original linkage involving descendants of n_a or n_b to the leaf nodes in ST_2.

Step 2: Visit all leaf nodes in ST_2 that are linked with both n_a and n_b to compute the second term in Equation (7.7).

Step 3: Aggregate the simweights on the leaf nodes to those nodes on level-1. Then further aggregate simweights to nodes on level-2, and so on.

Step 4: For each node n_i in ST_2 linked with n_a, and each sibling of n_i that is linked with n_b (we call it n_j), add $sw_{n_a n_i} \times \langle s(n_i, n_j), 1\rangle \times sw_{n_b n_j}$ to the first term of equation (7.7).

Suppose n_a is linked with m leaf nodes in ST_2, and n_b is linked with $O(m \cdot c)$ ones. It is easy to verify that the above procedure takes $O(mc \log_c N)$ time.

7.5.4 Iterative Adjustment of SimTrees

After building the initial SimTrees as described in Section 7.5.1, LinkClus needs to iteratively adjust both the similarities and structure of each SimTree. In Section 7.5.3 we have described how to compute the similarity between two nodes using similarities between their neighbor leaf nodes in other SimTrees. In this section we will introduce how to restructure a SimTree so that similar nodes are put together.

The structure of a SimTree is represented by the parent-child relationships, and such relationships may need to be modified in each iteration because of the modified similarities. In each iteration, for each node n, LinkClus computes n's linkage-based similarity with $parent(n)$ and the siblings of $parent(n)$. If n has higher similarity with a sibling node $\check{n}$ of $parent(n)$, then n will become a child of $\check{n}$, if $\check{n}$ has less than c children. The moves of low-level nodes can be considered as local adjustments on the relationships between objects, and the moves of high-levels nodes as global adjustments on the relationships between large groups of objects. Although each node can only be moved within a small range (*i.e.*, its parent's siblings), with the effect of both local and global adjustments, the tree restructure is often changed significantly in an iteration.

The procedure for restructuring a SimTree is shown in Algorithm 5. LinkClus tries to move each node n to be the child of a parent node that is most similar to n. Because each non-leaf node $\check{n}$ can have at most c children, if there are more than c nodes that are most similar to $\check{n}$, only the top c of them can become children of $\check{n}$, and the remaining ones are reassigned to be children of other nodes similar to them.

After restructuring a SimTree ST, LinkClus needs to compute the value associated with every edge in ST. For each edge between two sibling nodes, their similarity is directly computed as in Section 7.5.3. For each edge between a node n and its parent, LinkClus needs to compute the average similarity between n and all leaf nodes except descendants of $parent(n)$, and that for $parent(n)$. It can be proved that the average

Algorithm 5 Restructure SimTree

Input: a SimTree ST to be restructured, which is linked with SimTrees $ST_1, \ldots, ST_k$.

Output: The restructured ST.

```
$S_c \leftarrow$ all nodes in $ST$ except root //$S_c$ contains all child nodes
$S_p \leftarrow$ all non-leaf nodes in $ST$    //$S_p$ contains all parent nodes
for each node $n$ in $S_c$    //find most similar parent node for $n$
    for each sibling node $n'$ of $parent(n)$ (including $parent(n)$)
        compute $sim_l(n, n')$ using $ST_1, \ldots, ST_k$
    end
    sort the set of $sim_l(n, n')$ for $n$
    $p^*(n) \leftarrow n'$ with maximum $sim_l(n, n')$
end
while($S_c \neq \emptyset$)
    for each node $\check{n} \in S_p$    //assign children to $\check{n}$
        $q^*(\check{n}) \leftarrow \{n | p^*(n) = \check{n}\}$
        if $|q^*(\check{n})| < c$
        then $children(\check{n}) \leftarrow q^*(\check{n})$
        else
            $children(\check{n}) \leftarrow c$ nodes in $q^*(\check{n})$ most similar to $\check{n}$
            $S_p \leftarrow S_p - \{\check{n}\}$
        end
        $S_c \leftarrow S_c - children(\check{n})$
    end
    for each node $n \in S_c$
        $p^*(n) \leftarrow n'$ with maximum $sim_l(n, n')$ and $n' \in S_p$
    end
end
return $ST$
```

Figure 7.13: Algorithm *Restructure* SimTree

linkage-based similarity between n and all leaf nodes in ST except descendants of a non-leaf node n' is

$$\frac{sum_{ST_k}(n, root(ST)) - sum_{ST_k}(n, n'))}{weight_{ST_k}(n, root(ST)) - weight_{ST_k}(n, n'))} \tag{7.8}$$

Please note that equation (7.8) uses notations in Definition 17. With equation (7.8) we can compute the similarity ratio associated with each edge between a node and its parent. This finishes the computation of the restructured SimTree.

7.5.5 Complexity Analysis

In this section we analyze the time and space complexity of LinkClus. For simplicity, we assume there are two object types, each having N objects, and there are M linkages between them. Two SimTrees ST_1 and ST_2 are built for them. If there are more object types, the similarity computation between each pair of linked types can be done separately.

When a SimTree is built, LinkClus limits the number of nodes at each level. Suppose there are N_l nodes on level-l. The number of nodes on level-$(l+1)$ must be between $\frac{N_l}{c}$ and $\frac{\alpha N_l}{c}$ ($\alpha \in [1, 2]$ and usually $c \in [10, 20]$). Thus the height of a SimTree is $O(\log_c N)$.

In each iteration, LinkClus restructures each SimTree using similarities between nodes in the other SimTree, and then updates the values associated with edges in each SimTree. When restructuring ST_1, for each node n in ST_1, LinkClus needs to compute its similarity to its parent and parent's siblings, which are at most c nodes. Suppose n is linked with m leaf nodes in ST_2. As shown in Section 7.5.3, it takes $O(mc \log_c N)$ time to compute the n's similarity with its parent or each of its parent's siblings. Thus it takes $O(mc^2 \log_c N)$ time to compute the similarities between n and these nodes.

There are N leaf nodes in ST_1, which have a total of M linkages to all leaf nodes in ST_2. In fact all nodes on each level in ST_1 have M linkages to all leaf nodes in ST_2, and there are $O(\log_c N)$ levels. Thus it takes $O(Mc^2(\log_c N)^2)$ time in total to compute the similarities between every node in ST_1 and its parent and parent's siblings.

In the above procedure, LinkClus processes nodes in ST_1 level by level. When processing the leaf nodes, only the simweights of linkages involving leaf nodes and nodes on level-1 of ST_1 are attached to nodes in ST_2. There are $O(M)$ such linkages, and the simweights on the leaf nodes in ST_2 require $O(M)$ space. In ST_2 LinkClus only compares the simweights of sibling nodes, thus it can also process the nodes level by level. Therefore, the above procedure can be done in $O(M)$ space. Each SimTree has $O(N)$ nodes, and it takes $O(c)$ space to store the similarity between each node and its siblings (and its parent's siblings). Thus the space requirement is $O(M + Nc)$.

It can be easily shown that the procedure for restructuring a SimTree (Algorithm 1) takes $O(Nc)$ space and $O(Nc \log c)$ time, which is much faster than computing similarities.

After restructuring SimTrees, LinkClus computes the similarities between each node and its siblings. This can be done using the same procedure as computing similarities between each node and its parent's siblings. Therefore, each iteration of LinkClus takes $O(Mc^2(\log_c N)^2)$ time and $O(M + Nc)$ space. This is affordable for very large databases.

7.6 Empirical Study

In this section we report experiments to examine the efficiency and effectiveness of LinkClus. LinkClus is compared with the following approaches: (1) *SimRank* [62], an approach that iteratively computes pairwise similarities between objects; (2) *ReCom* [105], an approach that iteratively clusters objects using the cluster labels of linked objects; (3) SimRank with fingerprints [43] (we call it *F-SimRank*), an approach that pre-computes a large sample of random paths from each object and uses the samples of two objects to estimate their SimRank similarity; (4) SimRank with pruning (we call it *P-SimRank*) [62], an approach that approximates SimRank by only computing similarities between pairs of objects reachable within a few links.

SimRank and F-SimRank are implemented strictly following their papers. (We use decay factor 0.8 for F-SimRank, which leads to highest accuracy in DBLP database.) ReCom is originally designed for handling web queries, and contains a reinforcement clustering approach and a method for determining authoritativeness of objects. We only implement the reinforcement clustering method, because it may not be appropriate to consider authoritativeness in clustering. Since SimRank, F-SimRank and P-SimRank only provide similarities between objects, we use CLARANS [88], a k-medoids clustering approach, for clustering using such similarities. CLARANS is also used in ReCom since no specific clustering method is discussed in [105]. We compare LinkClus using both hierarchical clustering and CLARANS.

All experiments are performed on an Intel PC with a 3.0GHz P4 processor, 1GB memory, running Windows XP Professional. All approaches are implemented using Visual Studio.Net (C#). In LinkClus, α is set to $\sqrt{2}$. We will discuss the influences of c (max number of children of each node) on accuracy and efficiency in the experiments.

7.6.1 Evaluation Measures

Validating clustering results is crucial for evaluating approaches. In our test databases there are predefined class labels for certain types of objects, which are consistent with our clustering goals. Jaccard coefficient [101] is a popular measure for evaluating clustering results, which is the number of pairs of objects in same cluster and with same class label, over that of pairs of objects either in same cluster or with same class label. Because an object in our databases may have multiple class labels but can only appear in one cluster, there may be many more pairs of objects with same class label than those in same cluster. Therefore we use a variant of Jaccard coefficient. We say two objects are correctly clustered if they share at least one common class label. The accuracy of clustering is defined as the number of object pairs that are correctly clustered over that of object pairs in same cluster. Higher accuracy tends to be achieved when number of clusters is larger. Thus we let each approach generate the same number of clusters.

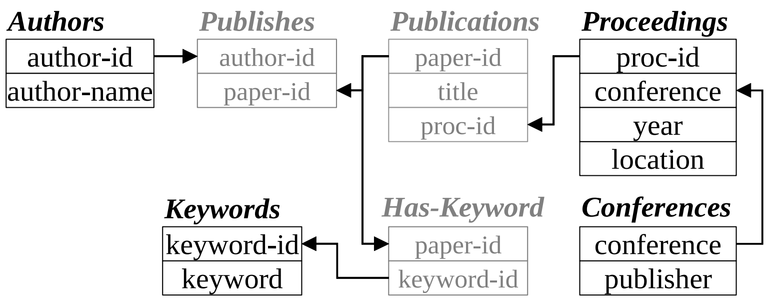


Figure 7.14: The schema of the DBLP database

7.6.2 DBLP Database

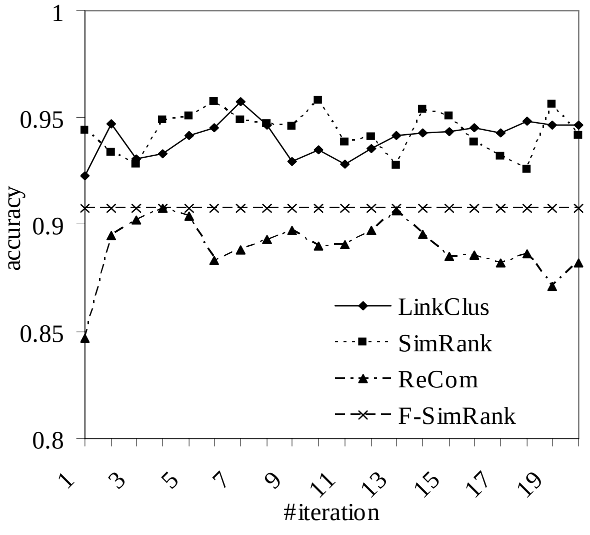
We first test on the DBLP database, whose schema is shown in Figure 7.14. It is extracted from the XML data of DBLP [32]. We want to focus our analysis on the productive authors and well-known conferences[1], and group them into clusters so that each cluster of authors (or conferences) are in a certain research area. We first select conferences that have been held for at least 8 times. Then we remove conferences that are not about computer science or are not well known, and there are 154 conferences left. We select 4170 most productive authors in those conferences, each having at least 12 publications. The *Publications* relation contains all publications of the selected authors in the selected conferences. We select about 2500 most frequent words (stemmed) in titles of the publications, except 50 most frequent words which are removed as stop words. There are four types of objects to be clustered: 4170 authors, 2517 proceedings, 154 conferences, and 2518 keywords. Publications are not clustered because too limited information is known for them (about 65% of publications are associated with only one selected author). There are about 100K linkages between authors and proceedings, 363K linkages between keywords and authors, and 363K between keywords and proceedings. Because we focus on clustering with linkages instead of keywords, we perform experiments both with and without the keywords.

We manually label the areas of the most productive authors and conferences to measure clustering accuracy. The following 14 areas are considered: Theory, AI, operating system, database, architecture, programming languages, graphics, networking, security, HCI, software engineering, information retrieval, bioinformatics, and CAD. For each conference, we study its historical call for papers to decide its area. 90% of conferences are associated with a single area. The other 10% are associated with multiple areas, such as KDD (database and AI). We analyze the research areas of 400 most productive authors. For each of them, we find her home page and infer her research areas from her research interests. If no research interests are

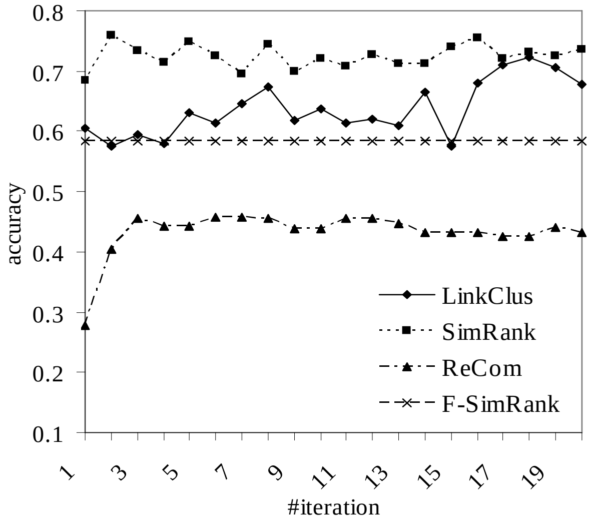
[1]Here conferences refer to conferences, journals and workshops. We are only interested in productive authors and well-known conferences because it is easier to determine the research fields related to each of them, from which the accuracy of clustering will be judged.

specified, we infer her research areas from her publications. On average each author is interested in 2.15 areas. In the experiments each type of objects are grouped into 20 clusters, and the accuracy is tested based on the class labels.

We first perform experiments without keywords. 20 iterations are used for SimRank, P-SimRank, ReCom, and LinkClus[2] (not including the initialization process of each approach). In F-SimRank we draw a sample of 100 paths (as in [43]) of length 20 for each object, so that F-SimRank can use comparable information as SimRank with 20 iterations. The accuracies of clustering authors and conferences of each approach are shown in Figure 7.15 (a) and (b), in which the x-axis is the index of iterations.



a) DBLP.Authors



b) DBLP.Conferences

Figure 7.15: Accuracy on DBLP w/o keywords

From Figure 7.15 one can see that SimRank is most accurate, and LinkClus achieves similar accuracy as SimRank. The accuracies of ReCom and F-SimRank are significantly lower. The error rates (*i.e.*, 1 – accuracy) of ReCom and F-SimRank are about twice those of SimRank and LinkClus on authors, and 1.5 times those of them on conferences. One interesting observation is that more iterations do not necessarily lead to higher accuracy. This is probably because cluster labels are not 100% coherent with data. In fact this is common for iterative clustering algorithms.

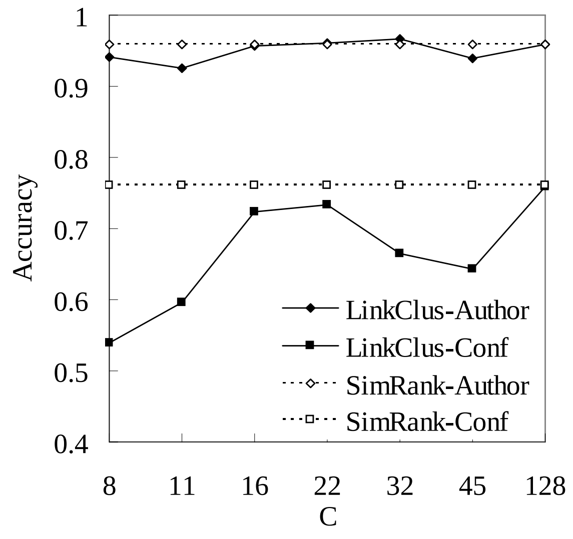
In the above experiment, LinkClus generates 20 clusters directly from the SimTrees: Given a SimTree, it first finds the level in which the number of nodes is most close to 20. Then it either keeps merging the most similar nodes if the number of nodes is more than 20, or keeps splitting the node with most descendant

[2]Since no frequent patterns of conferences can be found using the proceedings linked to them, LinkClus uses authors linked with conferences to find frequent patterns of conferences, in order to build the initial SimTree for conferences.

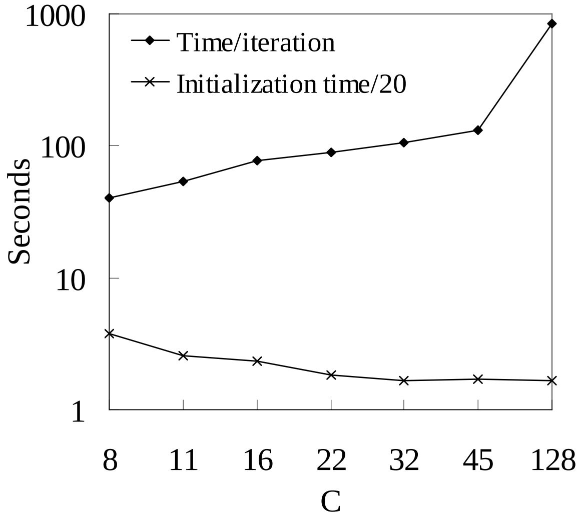
	Max accuracy		Time/iteration
	Authors	Conferences	
LinkClus	0.9574	0.7229	76.74 sec
LinkClus-Clarans	0.9529	0.7523	107.7 sec
SimRank	0.9583	0.7603	1020 sec
ReCom	0.9073	0.4567	43.1 sec
F-SimRank	0.9076	0.5829	83.6 sec

Table 7.1: Performances on DBLP without keywords

objects if otherwise, until 20 nodes are created. We also test LinkClus using CLARANS with the similarities indicated by SimTrees. The max accuracies and running time of different approaches are shown in Table 7.1. (The running time per iteration of F-SimRank is its total running time divided by 20.) One can see that the accuracy of LinkClus with CLARANS is slightly higher than that of LinkClus, and is close to that of SimRank. While SimRank is much more time consuming than other approaches.



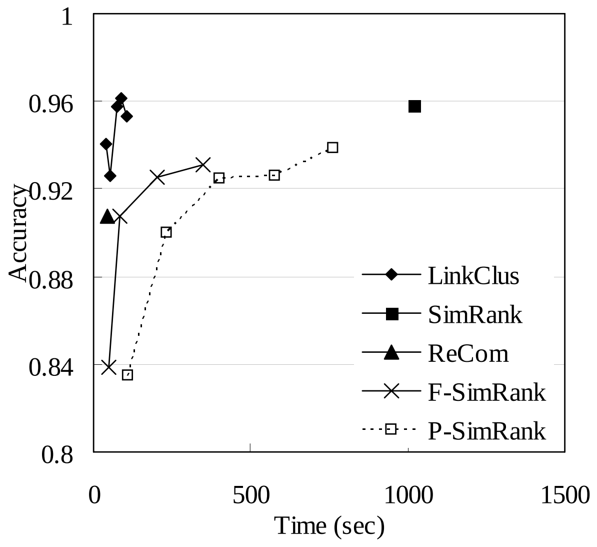
a) Max accuracy



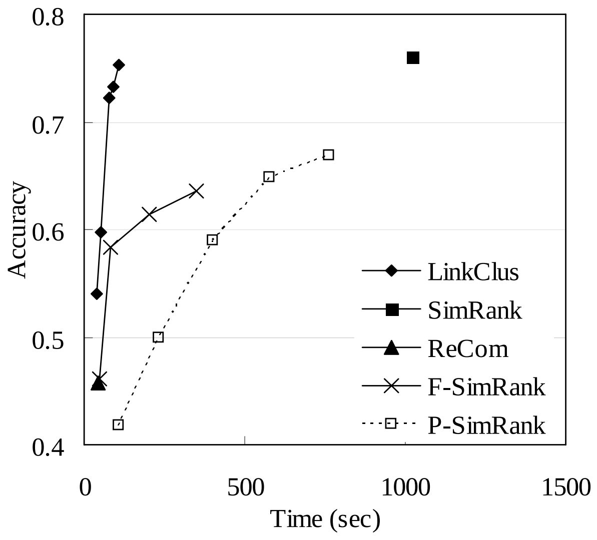
b) Running time

Figure 7.16: LinkClus with different c's

In the above experiments we use $c = 16$ for LinkClus. c is an important parameter for LinkClus, and thus we analyze its influences on accuracy and running time. As shown in Figure 7.16, we test c from 8 to 45, each differing by $\sqrt{2}$. The running time grows almost linearly, and the accuracy is highest when c equals 16 and 22. Theoretically, more similarities are computed and stored as c increases, and the accuracy should also increase (LinkClus becomes SimRank when c is larger than the number of objects). However, when c increases from 16 to 45, the SimTrees of authors and proceedings always have four levels, and that of conferences has three levels. Although more similarities are stored at lower levels of the SimTrees, there are a comparatively small number of nodes at the second highest level. Thus not much more information is stored when c increases from 16 to 45. On the other hand, it may be difficult to generate 20 clusters if the second highest level has too few nodes. In comparison, when $c = 128$ the SimTrees have less levels, and accuracies become higher. Figure 7.16 also shows that the initialization time of LinkClus is very insignificant.



a) DBLP.Authors

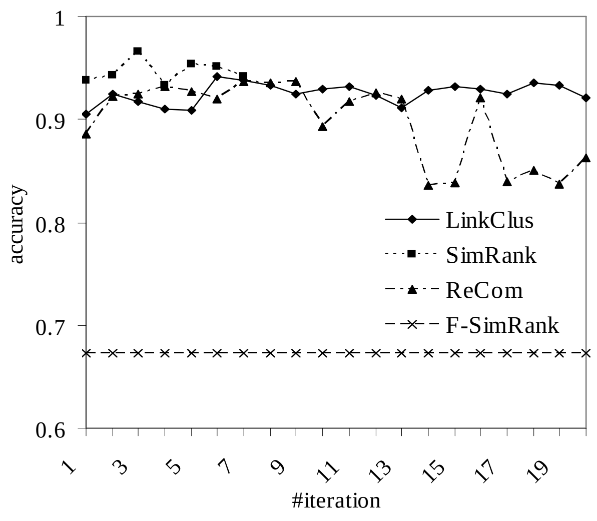


b) DBLP.Conferences

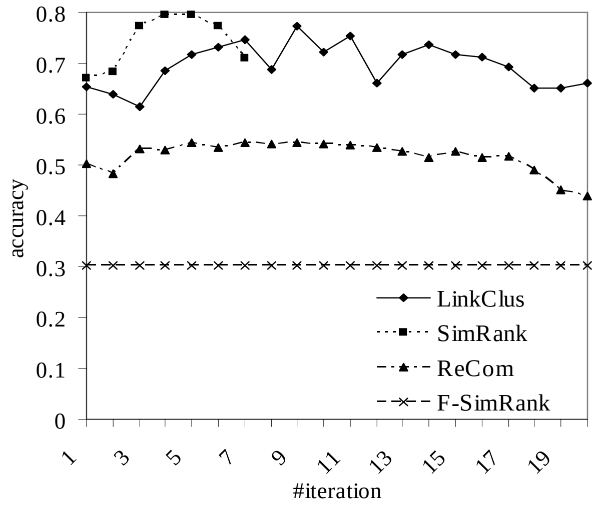
Figure 7.17: Accuracy vs. Time on DBLP w/o keywords

In many methods of linkage-based clustering there is a trade-off between accuracy and efficiency. This trade-off is shown in Figure 7.17, which contains the "accuracy vs. time" plots of SimRank, ReCom, LinkClus with different c's (8 to 22, including $c = 16$ with CLARANS), and F-SimRank with sample size of 50, 100, 200 and 400. It also includes SimRank with pruning (P-SimRank), which uses the following pruning method: For each object x, we only compute its similarity with the top-k objects that share most common neighbors with x within two links (k varies from 100 to 500). In these two plots, the approaches in the top-left region are good ones as they have high accuracy and low running time. It can be clearly seen that LinkClus greatly outperforms the other approaches, often in both accuracy and efficiency. In comparison, pruning technique of SimRank does not improve much on efficiency, because it requires using hashtables to store similarities, and an access to a hashtable is 5 to 10 times slower than that to a simple array.

Then we test on the DBLP database with keyword information. The accuracies of the approaches on authors and conferences are shown in Figure 7.18 (a) and (b). We only finish 7 iterations for SimRank because it is too time consuming. When using keywords, the accuracies of most approaches either increase



a) DBLP.Authors



b) DBLP.Conferences

Figure 7.18: Accuracy on DBLP with keywords

or remain the same. The only exception is F-SimRank, whose accuracy drops more than 20%. This is because there are many more linkages when keywords are used, but F-SimRank still uses the same sample size, which makes the samples much sparser.

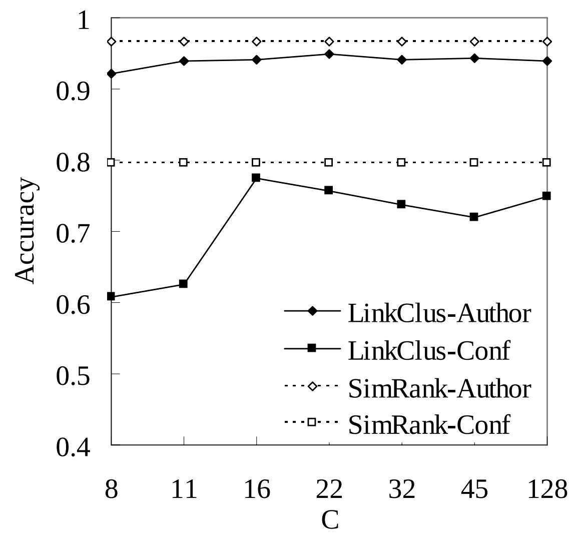
	Max accuracy		Time/iteration
	Authors	Conferences	
LinkClus	0.9412	0.7743	614.0 sec
LinkClus-Clarans	0.9342	0.7287	654.9 sec
SimRank	0.9663	0.7953	25348 sec
ReCom	0.9363	0.5447	101.2 sec
F-SimRank	0.6742	0.3032	136.3 sec

Table 7.2: Performances on DBLP with keywords

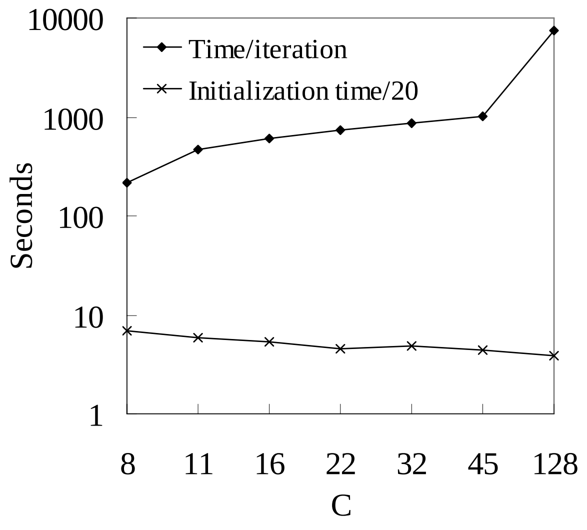
Table 7.2 shows the max accuracy and running time of each approach. The total number of linkages grows about 8 times when keywords are used. The running time of LinkClus grows linearly, and that of SimRank grows quadratically. Because ReCom considers linked clusters of each object, instead of linked objects, its running time does not increase too much. The running time of F-SimRank does not increase much as it uses the same number of random paths.

We also test LinkClus with different c's, as shown in Figure 7.19. The accuracy is highest when $c = 16$ or 22.

Figure 7.20 shows the curves of accuracies vs. running time (in log scale) of LinkClus, SimRank, ReCom, F-SimRank, and P-SimRank (100 to 400 similarity entries for each object). One can see that LinkClus is the

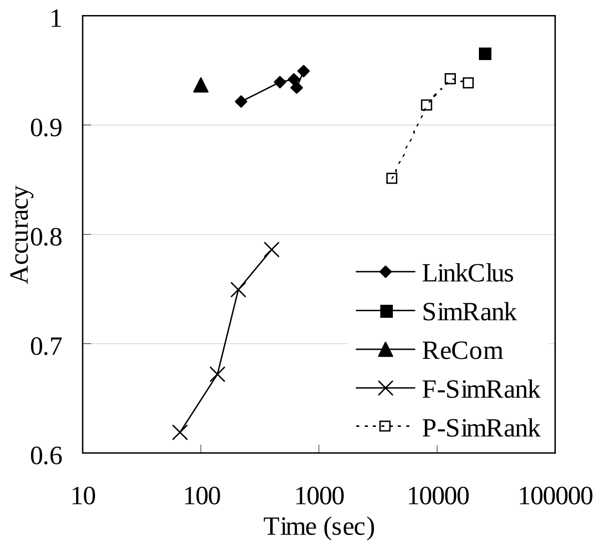


a) Max accuracy

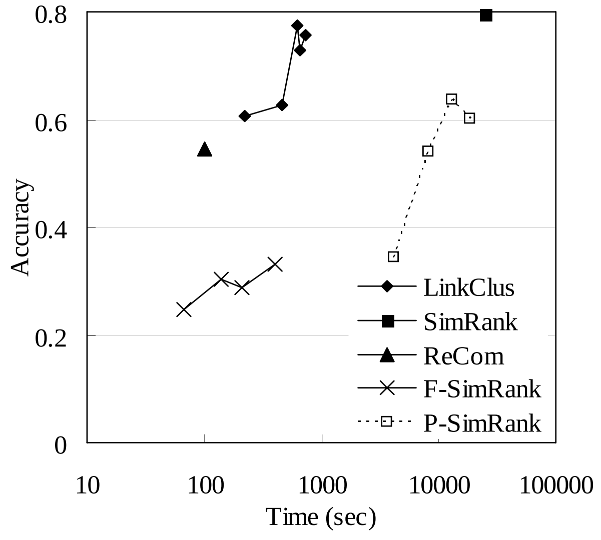


b) Running time

Figure 7.19: LinkClus with different c's



a) DBLP.Authors



b) DBLP.Conferences

Figure 7.20: Accuracy vs. Time on DBLP with keywords

most competitive method, as it achieves high accuracies and is 40 times more efficient than SimRank.

7.6.3 Email Dataset

We test the approaches on the Email dataset [89], which contains 370 emails on conferences, 272 on jobs, and 789 spam emails. To make the sizes of three classes more balanced, we keep all emails on conferences and jobs, and keep 321 spam emails. We select about 2500 most frequent words (stemmed). The database contains two types of objects, –emails and words, and about 141K linkages between them.

We use LinkClus ($c = 22$), SimRank, ReCom, F-SimRank (400 paths from each object), and the simple CLARANS algorithm to generate three clusters on emails and words. 20 iterations are used for LinkClus, SimRank, and ReCom. We do not test SimRank with pruning because this dataset is pretty small. Table 7.3 reports the max accuracy and running time of each approach. LinkClus achieves highest accuracy (slightly higher than that of SimRank), possibly because it captures the inherent hierarchy of objects.

	Max accuracy	Time
LinkClus	0.8026	78.98 sec/iteration
SimRank	0.7965	1958 sec/iteration
ReCom	0.5711	3.73 sec/iteration
F-SimRank	0.3688	479.7 sec
CLARANS	0.4768	8.55 sec

Table 7.3: Performances on Email dataset

7.6.4 Synthetic Databases

In this section we test the scalability and accuracy of each approach on synthetic databases. Figure 7.21 shows an example schema of a synthetic database, in which $R_1, R_2, R_3 R_4$ contain objects, and R_5, R_6, R_7, R_8 contain linkages. We use $RxTyCzSw$ to represent a database with x relations of objects, each having y objects which are divided into z clusters, and each object has w linkages to objects of another type (*i.e.*, selectivity is w). In each relation of objects R_i, the x objects are randomly divided into z clusters. Each cluster is associated with two clusters in each relation of objects linked with R_i. When generating linkages between two linked relations R_i and $R_{i\%4+1}$, we repeat the following procedure for $x \cdot w$ times: Randomly select an object o in R_i, and find the two clusters in $R_{i\%4+1}$ associated with the cluster of o. Then generate a linkage between o and a randomly selected object in these two clusters with probability $(1 - noise\_ratio)$, and generate a linkage between o and a randomly selected object with probability $noise\_ratio$. The default value of $noise\_ratio$ is 20%. It is shown in previous experiments that in most cases each approach can achieve almost the highest accuracy in 10 iterations, we use 10 iterations in this subsection. We let each approach generate z clusters for a database $RxTyCzSw$. For LinkClus we use $c = 16$ and do not use CLARANS.

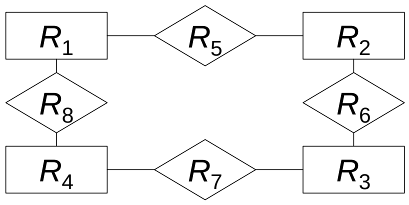
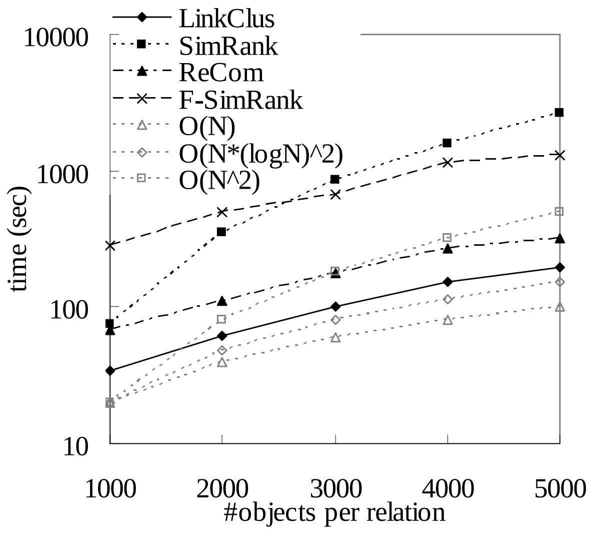
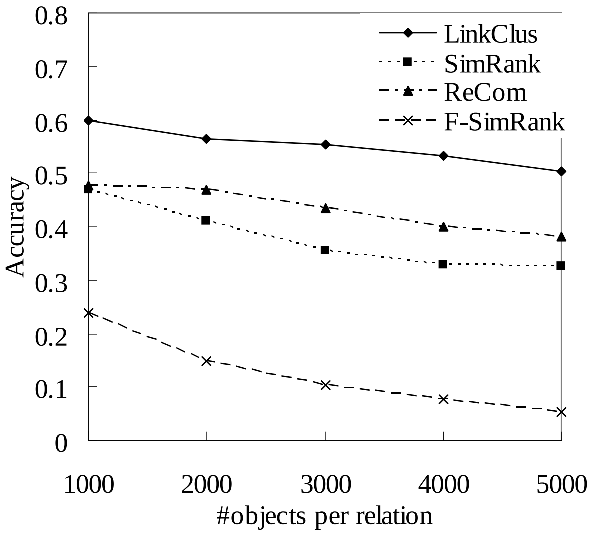


Figure 7.21: The schema of a synthetic database

We first test scalability w.r.t. the number of objects. We generate databases with 5 relations of objects, 40 clusters in each of them, and selectivity 10. The number of objects in each relation varies from 1000 to 5000. The running time and accuracy of each approach are shown in Figure 7.22. The time/iteration of F-SimRank is the total time divided by 10. With other factors fixed, theoretically the running time of LinkClus is $O(N(\log N)^2)$, that of *SimRank* is $O(N^2)$, and those of ReCom and F-SimRank are $O(N)$. We also show the trends of these bounds and one can see that the running time of the approaches are consistent with theoretical derivations. LinkClus achieves highest accuracy, followed by ReCom and then SimRank, and F-SimRank is least accurate. The possible reason for LinkClus and ReCom achieving high accuracy is that they group similar objects into clusters (or tree nodes) in the clustering process. Because clusters are clearly generated in data, using object groups in iterative clustering is likely to be beneficial.



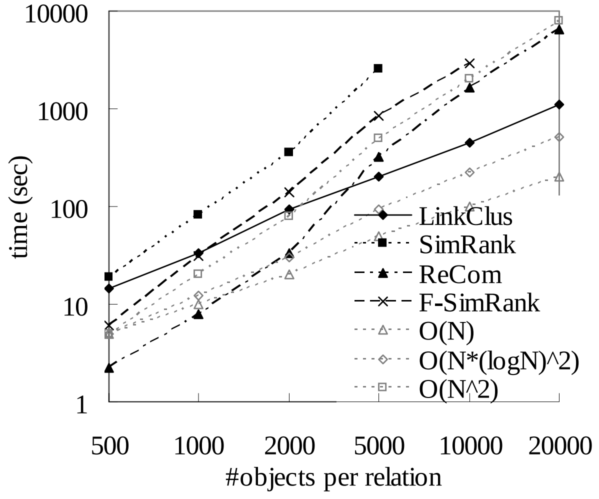
a) Time/iteration



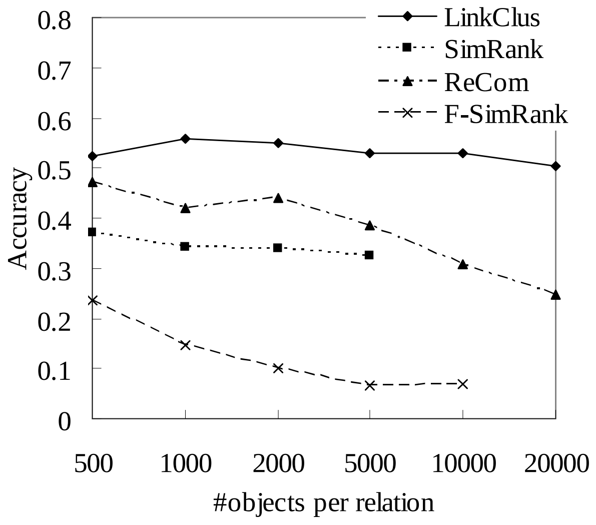
b) Accuracy

Figure 7.22: Performances on R5T*C40S10

In the last experiment the accuracy of each approach keeps decreasing as the number of objects increases. This is because the density of linkages decreases as cluster size increases. In R5T1000C40S10, each cluster has only 25 objects, each having 10 linkages to the two related clusters (50 objects) in other relations. In R5T5000C40S10, each cluster has 125 objects and the two related clusters have 250 objects, which makes the linkages much sparser. In the second experiment we increase the number of objects and clusters together to keep density of linkages fixed. Each cluster has 100 objects, and the number of objects per relation varies from 500 to 20000. In the largest database there are 100K objects and 1M linkages. The running time and accuracy of each approach are shown in Figure 7.23[3]. ReCom and F-SimRank are unscalable as their running time is proportional to the number of objects times the number of clusters, because they compute similarities between each object and each cluster medoid. The accuracies of LinkClus and SimRank do not change significantly, even the numbers of objects and clusters grow 40 times.



a) Time/iteration



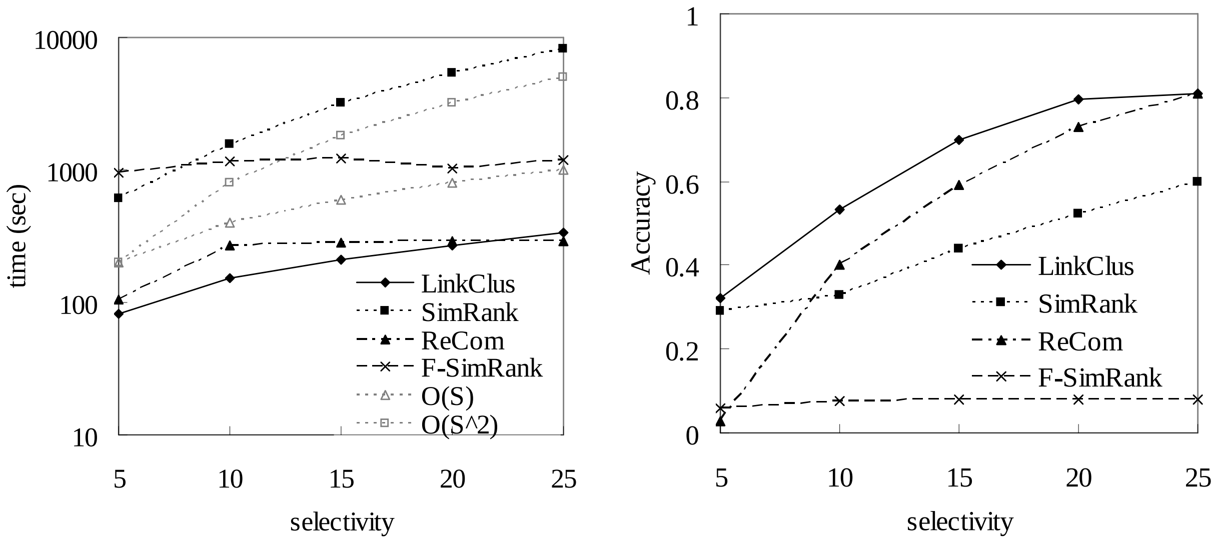
b) Accuracy

Figure 7.23: Performances on R5T*C*S10

Then we test each approach on databases with different selectivities, as shown in Figure 7.24. We generate databases with 5 relations of objects, each having 4000 objects and 40 clusters. The selectivity varies from 5 to 25. The running time of LinkClus grows linearly and that of SimRank quadratically with the selectivity, and those of ReCom and F-SimRank are only slightly affected. These are consistent with theoretical derivations. The accuracies of LinkClus, SimRank and ReCom increase quickly when selectivity increases, showing that density of linkages is crucial for accuracy. The accuracy of F-SimRank remains stable because it does not use more information when there are more linkages.

Finally we test the accuracy of each approach on databases with different noise ratios, as shown in Figure 7.25. We change noise ratio from 0 to 0.4. The accuracies of LinkClus, SimRank and F-SimRank decrease

[3]We do not test SimRank and F-SimRank on large databases because they consume too much memory.



a) Time/iteration b) Accuracy

Figure 7.24: Performances on R5T4000C40S*

with a stable rate when noise ratio increases. ReCom is most accurate when noise ratio is less than 0.2, but is least accurate when noise ratio is greater than 0.2. It shows that LinkClus and SimRank are more robust than ReCom in noisy environments.

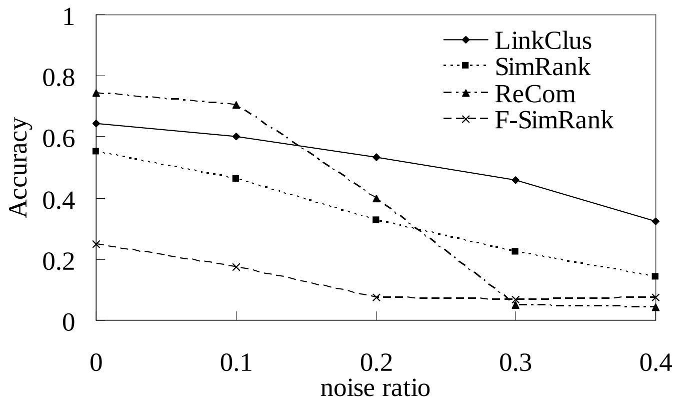


Figure 7.25: Accuracy vs. noise ratio on R5T4000C40S10

7.7 Summary

In this chapter we propose a highly effective and efficient approach of linkage-based clustering, LinkClus, which explores the similarities between objects based on the similarities between objects linked with them. We propose similarity-based hierarchical structure called SimTree as a compact representation for similarities, and propose an efficient algorithm for computing similarities, which avoiding pairwise computations by merging similarity computations that go through common paths. Experiments show LinkClus achieves high efficiency, scalability, and accuracy in clustering multi-typed linked objects.

Chapter 8

Conclusions and Future Directions

In the past decade data mining research has been focused on data in regular formats, such as individual tables and sets of transactions. However, most of the structured data in the world is stored in relational databases, and relational data can often provide crucial information for data mining tasks. For example, classification of an object depends on its neighbor objects in the database, and clustering depends on the relational linkages between objects.

Although relational databases provide rich information for relationships between objects, there is not much study on how to discover and utilize such relationships. One possible reason is that the complexity of relational data makes it difficult to design efficient and effective approaches. In a relational database each object has many neighbor objects, and two objects (or two relations) can often be connected in many different ways. Another interesting property is that, the relationship between two objects often depends on the relationships between their neighbor objects, which makes the analysis more complicated.

In this thesis we have studied how to apply data mining approaches on relational data. With multi-relational data mining, we can solve some problems that are much more difficult without capabilities of multi-relational analysis. For example, we can analyze the identity of objects, in order to detect duplicates or to merge identical objects with different names. We can also cluster objects that have similar neighbors (e.g., related to similar objects), but will be considered dissimilar without multi-relational analysis.

8.1 Summary of Contributions

In this thesis we present our solutions for performing several most important data mining tasks in multi-relational environments, including classification, clustering, duplicate detection, object distinction, and similarity analysis. In general, we made the following contributions in this thesis.

- **System framework**

  We propose a general methodology for multi-relational data mining. It includes *tuple ID propagation*, an efficient method for transferring information across different relations for effective cross-relational data mining. It also contains a heuristic method for searching for pertinent features in the relational database. Our methodology focuses on two types of information in the data mining process: multi-relational features (neighbor objects) and linkages between objects.

- **Multi-relational features and linkages**

  In this thesis we employ two types of information for data mining: multi-relational features and linkages between objects. A multi-relational feature is defined based on the neighbor objects in a certain relation, which models certain properties of objects. A linkage between two objects is a chain of objects that links the two objects together, which has a certain semantic meaning. These two types of information enables us to analyze the properties and relationships of objects.

- **Algorithms**

  We develop several core algorithms for several data mining tasks: (1) CrossMine for performing efficient multi-relational classification with tuple ID propagation, (2) CrossClus for multi-relational clustering with user guidance incorporated, (3) Relom for duplicate detection using multi-relational information, (4) DISTINCT for distinguishing objects with identical names in relational databases, and (5) LinkClus for analyzing similarities between objects using similarities between related objects. These algorithms provide highly efficient and effective solutions for the several most important data mining problems in relational environments.

- **Evaluation**

  We evaluate each proposed approach on both synthetic and real relational databases, and the experiments show that each approach achieves high accuracy, efficiency, and scalability.

8.2 Future Work in Multi-relational Data Mining

In the past several years, the research on multi-relational data mining has been significantly advanced. Based on analysis on these studies, I believe there are several important directions that should be explored to further advance this field.

- **Multi-relational data mining on new properties of data objects**

  In PageRank [92] by Page et al. and Authority-Hub Analysis [70] by Kleinberg, the authoritativeness of web pages (with or without respect to certain queries) are analyzed using the linkage information (hyperlinks), and effective web search techniques are proposed. In SimRank [62] by Jeh and Widom and our work LinkClus [113], the relational information is used to analyze the similarities between objects. These studies have solved problems that are very difficult without using multi-relational or linkage information, which illustrate the importance of relational and linkage data.

Besides authoritativeness and similarities, there are many other types of properties that could be explored with linkage information. For example, in the catalog database of an online retailer, we may use multi-relational data mining to identify related products, which can be recommended to customers when they browse products (e.g., the recommendations on Amazon.com). Two products can be considered as related if they share similar properties, or if they are linked to similar manufacturers, customers, or reviewers. In many applications such as recommending movies, books, music, etc., multi-relational data mining can be combined with collaborative filtering [17] to make accurate predictions. Compared with collaborative filtering, multi-relational data mining can utilize a much wider spectrum of information, including many types of linkages between many types of objects.

- **Data quality control with linkage data**

  Although the data volume on the internet is extremely high and is increasing rapidly, the quality of such data is not guaranteed. Some organizations have made significant efforts in building a trustable knowledge base using mass collaboration, among whom the most noticeable one is www.wikipedia.org. However, these knowledge bases only cover a very small portion of information on the web, and do not include newly updated information. For example, there are many online retailers selling all kinds of digital cameras, but most of them cannot provide the complete and correct specifications. Another example is that different online book retailers often provide different information for the same book (e.g., authors and number of pages).

  It is very desirable if one can build a system to reconcile the conflictive information about certain objects, which is provided by different web sites. This can be modeled as a linkage analysis problem, as there are linkages between objects, properties of objects, and information providers (e.g., web sites). This can also be viewed as a mass collaboration problem, because the correct information is likely to be acquired by many different information providers. But it is also a very challenging problem since the incorrect information can also be transmitted between different information providers.

8.3 Envisioning A Unified Multi-relational Data Mining System

As can be seen from the discussions above, there are many applications in multi-relational data mining, such as classification, clustering, similarity analysis, duplicate detection, etc. These applications can significantly influence each other. For example, similarity analysis is of crucial importance for clustering. Another example is that, duplicate detection can help other applications by merging duplicate entries, while clustering and similarity analysis play essential roles in duplicate detection.

Based the above observation, we believe it is highly desirable to build a unified multi-relational data mining system, which can integrate different modules and can use the output of one module as the input of another module. For example, the similarities generated by similarity analysis can be used in clustering analysis, while the clusters generated can be feeded back to the module of similarity analysis as background information.

The greatest challenge in building a unified multi-relational data mining system is to build a unified interface for different data mining modules. Different modules must work on the same format of relational, tabular, and linkage data. Each module must generate output that can be understood by other modules. Another challenge is that, some information (e.g., similarities between objects) may be shared by multiple modules, and there should be an efficient method to store and share information among different modules. This unified system should also have good extensibility, so that new modules can be easily invented and integrated into the system.

References

[1] C. C. Aggarwal, C. Procopiuc, J. L. Wolf, P. S. Yu, and J. S. Park. Fast Algorithms for Projected Clustering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*, Philadelphia, Pennsylvania, June 1999.

[2] C. C. Aggarwal, and P. S. Yu. Finding Generalized Projected Clusters in High Dimensional Spaces. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, Dallas, Texas, May 2000.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, Washington, D.C., May 1993.

[4] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, August 2002.

[5] A. Appice, M. Ceci, and D. Malerba. Mining model trees: a multi-relational approach. In *Proceedings of 13rd International Conference on Inductive Logic Programming*, Szeged, Hungary, September 2003.

[6] J. M. Aronis and F. J. Provost. Increasing the Efficiency of Data Mining Algorithms with Breadth-First Marker Propagation. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD'97)*, Newport Beach, California, August 1997.

[7] R. Bekkerman, R. El-Yaniv, and A. McCallum. Multi-way distributional clustering via pairwise interactions. In *Proceedings of the 22nd International Conference on Machine Learning (ICML'05)*, Bonn, Germany, August 2005.

[8] I. Bhattacharya and L. Getoor. Iterative Record Linkage for Cleaning and Integration. In *Proceedings of the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'04)*, Paris, France, June 2004.

[9] I. Bhattacharya and L. Getoor. Relational Clustering for Multi-type Entity Resolution. In *Proceedings of the 4th Workshop on Multi-Relational Data Mining (MRDM'05)*, Chicago, Illinois, August 2005.

[10] M. Bilenko, S. Basu, and R. J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *Proceedings of the 21st International Conference (ICML'04)*, Banff, Alberta, Canada, July 2004.

[11] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, Washington, DC, August 2003.

[12] H. Blockeel, L. Dehaspe, B. Demoen, et al. Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. In *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

[13] H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of logical decision trees. In *Proceedings of the 15th International Conference on Machine Learning (ICML'98)*, Madison, WI, July 1998.

[14] H. Blockeel, L. De Raedt, N. Jacobs, and B. Demoen. Scaling up inductive logic programming by learning from interpretations. In *Data Mining and Knowledge Discovery*, 3(1):59–93, 1999.

[15] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. In *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

[16] J. Bockhorst, M. Craven, D. Page, J. Shavlik, and J. Glasner. A Bayesian network approach to operon prediction. In *Bioinformatics*, 19(10):1227–1235, 2003.

[17] J. S. Breese, D. Heckerman, C. Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. Technical Report MSR-TR-98-12, Microsoft Research, 1998.

[18] R. Brause, T. Langsdorf, M. Hepp. Neural Data Mining for Credit Card Fraud Detection. In *Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'99)*, Chicago, Illinois, November 1999.

[19] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. In *Data Mining and Knowledge Discovery*, 2:121–168, 1998.

[20] D. Chakrabarti, S. Papadimitriou, D.S. Modha, and C. Faloutsos. Fully automatic cross-associations. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'04)*, Seattle, Washington, August 2004.

[21] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, San Diego, California, June 2003.

[22] P. Cheeseman, et al. AutoClass: A Bayesian Classfication System. In *Proceedings of the 5th International Conference on Machine Learning (ICML'88)*, Ann Arbor, Michigan, June 1988.

[23] Y. Cheng, and G. M. Church. Biclustering of expression data. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB'00)*, San Diego, California, 2000.

[24] P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Proceedings of the European Working Session on Learning*, Porto, Portugal, March 1991.

[25] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, Seattle, Washington, June 1998.

[26] CS Dept Dataset. http://dm1.cs.uiuc.edu/csuiuc_dataset/

[27] C. L. Curotto, N. F. F. Ebecken, and H. Blockeel. Multi-relational data mining in Microsoft SQL Server 2005. In *Proceedings of 7th International Conference on Data, Text and Web Mining and their Business Applications and Management Information Engineering*, Prague, Czech Republic, July 2006.

[28] L. Dehaspe and L. De Raedt. Mining Association Rules in Multiple Relations. In *Proceedings of the 7th International Workshop on Inductive Logic Programming (ILP'97)*, Prague, Czech, September 1997.

[29] L. Dehaspe and H. Toivonen. Discovery of relational association rules. In *Relational Data Mining*, Springer-Verlag, 2001.

[30] L. Dehaspe, H. Toivonen, and R. King. Finding frequent substructures in chemical compounds. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD'98)*, New York, New York, August 1998.

[31] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, Washington, D.C., August 2003.

[32] DBLP Bibliography. http://www.informatik.uni-trier.de/∼ley/db/

[33] C. Ding, and H. Peng. Minimum Redundancy Feature Selection from Microarray Gene Expression Data. In *Proceedings of the 2nd IEEE Computer Society Bioinformatics Conference (CSB'03)*, Stanford, California, August 2003.

[34] A. Doan, Y. Lu, Y. Lee, and J. Han. Object Matching for Information Integration: A Profiler-Based Approach. In *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb'03)*, Acapulco, Mexico, August 2003.

[35] P. Domingos. Prospects and challenges for multi-relational data mining. In *ACM SIGKDD Explorations Newsletter*, 5(1):80–83, 2003.

[36] X. Dong, A. Halevy, and J. Madhavan. Reference Reconciliation in Complex Information Spaces. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*, Baltimore, Maryland, June 2005.

[37] J. G. Dy and C. E. Brodley. Feature Selection for Unsupervised Learning. In *Journal of Machine Learning Research*, 5:845–889, 2004.

[38] S. Dzeroski. Inductive logic programming and knowledge discovery in databases. In *Advances in Knowledge Discovery and Data Mining*, pp. 117C152. AAAI Press, 1996.

[39] W. Emde and D. Wettschereck. Relational Instance-Based Learning. In *Proceedings of the 13th International Conference on Machine Learning (ICML'96)*, Bari, Italy, July 1996.

[40] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'99)*, Cambridge, Massachusetts, August 1999.

[41] T. Fawcett and F. Provost. Adaptive Fraud Detection. In *Data Mining and Knowledge Discovery*, 1(3):291–316, 1997.

[42] I. Felligi and A. Sunter. A theory for record linkage. In *Journal of the American Statistical Society*, 64:1183–1210, 1969.

[43] D. Fogaras and B. Rácz. Scaling link-base similarity search. In *Proceedings of the 14th international conference on World Wide Web (WWW'05)*, Chiba, Japan, May 2005.

[44] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, Stockholm, Sweden, July 1999.

[45] V. Ganti, J. Gehrke, and R. Ramakrishnan. CACTUS - Clustering Categorical Data Using Summaries. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'99)*, San Diego, California, August 1999.

[46] T. Gärtner, J. W. Lloyd, and P. A. Flach. Kernels and Distances for Structured Data. In *Machine Learning*, 57(3): 205–232, 2004.

[47] H. Garcia-Molina, J. D. Ullman, and J. Widom. In *Database Systems: The Complete Book.* Prentice Hall, 2002.

[48] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest: A framework for fast decision tree construction of large datasets. In *Proceedings of 24th International Conference on Very Large Data Bases (VLDB'98)*, New York, New York, August 1998.

[49] L. Getoor. Link Mining: A New Data Mining Challenge. In *SIGKDD Explorations*, 5(1):84–89, 2003.

[50] L. Getoor and M. Sahami. Using probabilistic relational models for collaborative filtering. In *Proceedings of the 1999 International WEBKDD Workshop*, San Diego, California, August 1999.

[51] L. Getoor, B. Taskar, and D. Koller. Selectivity Estimation using Probabilistic Models. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, Santa Barbara, California, May 2001.

[52] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, Roma, Italy, September 2001.

[53] S. Guha, R. Rastogi, and K. Shim. CURE: an efficient clustering algorithm for large databases. In *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, Seattle, Washington, June 1998.

[54] I. Guyon and A. Elisseeff. An Introduction to Variable and Feature Selection. In *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[55] M. A. Hall. Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning. In *Proceedings of the 17th International Conference on Machine Learning (ICML'00)*, Standord, California, June 2000.

[56] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[57] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, Dallas, Texas, May 2000.

[58] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, Maebashi City, Japan, December 2002.

[59] M. A. Hernandez and S. J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, San Jose, California, May 1995.

[60] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, August 2002.

[61] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. In *ACM Computing Surveys*, 31:264–323, 1999.

[62] G. Jeh and J. Widom. SimRank: A measure of structural-context similarity. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)*, Edmonton, Alberta, Canada, August 2002.

[63] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA'03)*, Kyoto, Japan, March 2003.

[64] D. V. Kalashnikov, S. Mehrotra, and Z. Chen. Exploiting Relationships for Domain-Independent Data Cleaning. In *Proceedings of the 5th SIAM International Conference on Data Mining (SDM'05)*, Newport Beach, California, April 2005.

[65] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis*. John Wiley and Sons, 1990.

[66] H. Kim and S. Lee. A semi-supervised document clustering technique for information organization. In *Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management (CIKM'00)*, McLean, Virginia, November 2000.

[67] M. Kirsten and S. Wrobel. Relational Distance-Based Clustering. In *Proceedings of the 8th International Workshop Inductive Logic Programming (ILP'98)*, Madison, Wisconsin, July 1998.

[68] M. Kirsten and S. Wrobel. Extending K-Means Clustering to First-order Representations. In *Proceedings of the 10th International Workshop Inductive Logic Programming (ILP'00)*, London, U.K., July 2000.

[69] D. Klein, S. D. Kamvar, and C. Manning. From Instance-level Constraints to Space-Level Constraints: Making the Most of Prior Knowledge in Data Clustering. In *Proceedings of the 19th International Conference Machine Learning (ICML'02)*, Sydney, Australia, July 2002.

[70] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Journal of the ACM*, 46(5):604–632, 1999.

[71] S. Kok and P. Domingos. Learning the Structure of Markov Logic Networks. In *Proceedings of the 22nd International Conference Machine Learning (ICML'05)*, Bonn, Germany, August 2005.

[72] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference Machine Learning (ICML'01)*, Williamstown, Massachusetts, June 2001.

[73] N. Lavrac and S. Dzeroski. In *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[74] D. Liben-Nowell and J. M. Kleinberg. The link prediction problem for social networks. In *Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management (CIKM'03)*, New Orleans, Louisiana, November 2003.

[75] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the 5th Berkeley Symposium on Mathematics Statistic and Probability*, Berkeley, California, 1967.

[76] A. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00)*, Boston, Massachusetts, August 2000.

[77] T. M. Mitchell. In *Machine Learning.* McGraw Hill, 1997.

[78] P. Mitra, C. A. Murthy, and S. K. Pal. Unsupervised Feature Selection Using Feature Similarity. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(3):301–312, 2002.

[79] R. Motwani and P. Raghavan. In *Randomized Algorithms.* Cambridge University Press, 1995.

[80] S. Muggleton. In *Inductive Logic Programming.* Academic Press, New York, NY, 1992.

[81] S. Muggleton. Inverse entailment and progol. In *New Generation Computing, Special issue on Inductive Logic Programming*, 13:245–286, 1995.

[82] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First International Workshop on Algorithmic Learning Theory (ALT'90)*, Tokyo, Japan, October 1990.

[83] J. Neville, D. Jensen, L. Friedland, and M. Hay. Learning Relational Probability Trees. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, Washington, DC, August 2003.

[84] J. Neville, O. Simsek, D. Jensen, J. Komoroske, K. Palmer, and H. Goldberg. Using relational knowledge discovery to prevent securities fraud. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'05)*, Chicago, Illinois, August 2005.

[85] H. Newcombe, J. Kennedy, S. Axford, and A. James. Automatic linkage of vital records. In *Science*, 130:954–959, 1959.

[86] M. E. J. Newman. The structure and function of complex networks. In *SIAM Review*, 45(2):167–256, 2003.

[87] J. Newton and R. Greiner. Hierarchical probabilistic relational models for collaborative filtering. In *Proceedings of the ICML 2004 Workshop on Statistical Relational Learning and its Connections to Other Fields*, Banff, Alberta, Canada, July 2004.

[88] R. T. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, Santiago de Chile, Chile, September 1994.

[89] F. Nielsen. Email dataset. http://www.imm.dtu.dk/ ~rem/data/Email-1431.zip.

[90] S. Nijssen and J. N. Kok. Efficient frequent query discovery in Farmer. In *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'03)*, Cavtat-Dubrovnik, Croatia, September 2003.

[91] D. Page and M. Craven. Biological applications of multi-relational data mining. In *ACM SIGKDD Explorations Newsletter*, 5(1):69–79, 2003.

[92] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[93] PKDD'99 Discovery Challenge. http://lisp.vse.cz/pkdd99/Challenge/chall_prog.htm

[94] A. Popescul, L. Ungar, S. Lawrence, and M. Pennock. Towards structural logistic regression: Combining relational and statistical learning. In *Proceedings of the First Multi-Relational Data Mining Workshop (MRDM'02)*, Alberta, Canada, August 2002.

[95] J. R. Quinlan. In *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[96] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proceedings of the 1993 European Conference on Machine Learning (ECML'93)*, Vienna, Austria, April 1993.

[97] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)*, Edmonton, Alberta, Canada, August 2002.

[98] R. Sibson. SLINK: An optimally efficient algorithm for the single-link cluster method. In *The Computer Journal*, 16(1):30-34, 1973.

[99] P. Singla and P. Domingos. Object Identification with Attribute-Mediated Dependences. In *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases*, Porto, Portugal, October 2005.

[100] A. Srinivasan, R. D. King, S. H. Muggleton, and M. Sternberg. The predictive toxicology evaluation challenge. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, Nagoya, Japan, August 1997.

[101] P. N. Tan, M. Steinbach, and W. Kumar. *Introdution to data mining*. Addison-Wesley, 2005.

[102] B. Taskar, E. Segal, and D. Koller. Probabilistic classification and clustering in relational data. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, Seattle, Washington, August 2001.

[103] K. Wagstaff, C. Cardie, S. Rogers, and S. Schroedl. Constrained k-means clustering with background knowledge. In *Proceedings of the 18th International Conference Machine Learning (ICML'01)*, Williamstown, Massachusetts, June 2001.

[104] J. Wang, J. Han, and J. Pei. CLOSET+: searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, Washington, D.C., August 2003.

[105] J. D. Wang, H. J. Zeng, Z. Chen, H. J. Lu, L. Tao, and W. Y. Ma. ReCoM: Reinforcement clustering of multi-type interrelated data objects. In *Proceedings of the 26th International Conference on Research and Development in Information Retrieval*, Toronto, Canada, July 2003.

[106] Y. Weiss. Correctness of Local Probability Propagation in Graphical Models with Loops. In *Neural Computation* 12(1):1–41, 2000.

[107] W. Winkler. The state of record linkage and current research problems. Technical Report, Statistical Research Division, U.S. Bureau of the Census, 1999.

[108] S. Wrobel. An algorithm for multi-relational discovery of subgroups. In *European Conf. Principles and Practice of Knowledge Discovery in Databases*, Trondheim, Norway, 1997.

[109] X. Wu and S. Zhang. Synthesizing high-frequency rules from different data sources. In *IEEE Transactions on Knowledge and Data Engineering*, 15(2):353-367, 2003.

[110] E. P. Xing, A. Y. Ng, M. I. Jordan, and S. Russell. Distance metric learning, with application to clustering with side-information. In *Advances in Neural Information Processing Systems 15 (NIPS'02)*, Vancouver, Canada, December 2002.

[111] X. Yin, J. Han, J. Yang, and P. S. Yu. CrossMine: Efficient Classification Across Multiple Database Relations. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, Boston, Massachusetts, March 2004.

[112] X. Yin, J. Han, and P. S. Yu. Cross-Relational Clustering with User's Guidance. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'05)*, Chicago, Illinois, August 2005.

[113] X. Yin, J. Han, and P. S. Yu. LinkClus: Efficient Clustering via Heterogeneous Semantic Links In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06)*, Seoul, Korea, September 2006.

[114] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, Montreal, Canada, June 1996.

Author's Biography

Xiaoxin Yin will be graduating in May 2007 with a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. He received his Master of Science in Computer Science from the University of Illinois at Urbana-Champaign in 2003, and his Bachelor of Engineering in Computer Science and Technology from Tsinghua University in 2001.

He will be joining Google Inc. after finishing his Ph.D. degree. His research interests include data mining and link analysis in multi-relational databases, and their applications in different disciplines such as e-Commerce, web search, and bioinformatics.