UNIVERSITY OF KAISERSLAUTERN

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

A Framework for XML Similarity Joins

by

Leonardo Andrade Ribeiro

A thesis submitted in partial fulfillment of the requirements for the degree of Doktor der Ingenieurwissenschaften (Dr.-Ing.)

to the

Department of Computer Science

under supervision of

Prof. Dr.-Ing. Dr. h. c. Theo Härder.

DBIS Datenbanken und Informationssysteme

July 2010

A Framework for XML Similarity Joins

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

MSc. Leonardo Andrade Ribeiro

Dekan des Fachbereichs Informatik:

Prof. Dr. Karsten Berns

Promotionskommission:

Vorsitzender:	Prof. Dr. Klaus Madlener
Berichterstatter:	Prof. Dr. Dr. Theo Härder
	Prof. Dr. Michael Böhlen

Datum der wissenschaftlichen Aussprache:

15. July 2010

D 386

To my family

Acknowledgments

This thesis is the result of my work as doctoral candidate in the Database and Information Systems Group at the University of Kaiserslautern. In the course of this work, I have been fortunate to receive a great deal of support, advice, help, and encouragement of many people, and it is a pleasure to acknowledge my debt to them.

First, I would like to thank my family: my parents, Lázaro and Célia, my sisters, Keli and Kátia, my brother-in-law, Ziad, and my lovely nephews, Thomas and Lucas. Thank you for all the love, care, and encouragement along these years. This thesis is dedicated to you and would not have been possible without you.

Next, I would like to express my deep gratitude to my supervisor Prof. Dr. Dr. Theo Härder, for accepting me in his research group, for his generosity in advising me, frequently extending beyond his professional duties, and for keeping me motivated to push my own boundaries. It is always a bewilderment when I look back at all the things I have learned from him; clearly, I could not have been more successful in any other place. Working with Prof. Härder was a pleasure and an honor to me.

Special thanks are given to Prof. Dr. Aldo von Wangenheim, who was my Master supervisor, for nourishing my interest in research, encouraging me to challenging myself and embark on a PhD, and being always available for supporting me during the whole process. I would also like to thank Prof. Dr. Michael M. Richter, who was the doctoral supervisor of Prof. von Wangenheim. I was fortunate to have my office close to his and, thus, just a few steps from obtaining support and advice.

I would like to thank Prof. Dr. Michael Böhlen, whose work is cited many times in this thesis. In the early stages of my work, I was looking for ways to meet effectiveness and efficiency when evaluating structural similarity. As I found the paper of Prof. Böhlen on *pq*-gram similarity [ABG05], I was amazed by its elegance and technical depth and immediately adopted it as a reference for my subsequent work on structural similarity. Later, I was honored in having Prof. Böhlen in the assessment committee of my thesis. I would like to extend my thanks to the co-authors of the *pq*-grams paper as well: Prof. Dr. Nikolaus Augsten and Prof. Dr. Johann Gamper.

I would also like to thank Prof. Dr. Klaus Madlener and Prof. Dr. Stefan Deßloch, members of the PhD board, for their guidance in the first phase of the PhD program. Prof. Madlener also promptly accepted the role of chairman of the assessment committee and provided for a pleasant atmosphere during the defense of my thesis.

This work profited immensely from my participation in the XTC project. XTC provided me with a rock-solid and high-performance experimental testbed and a clean and well-rounded architecture to which I could plug in my "similarity stuff". Even more importantly, I benefited from many insightful and thought-provoking discussions with my colleagues during our regular project meetings. Thus, I would like to express my gratitude to all current and former members of the XTC team. In particular, special thanks go to Dr. Christian Mathis, who was the technical leader of XTC during most of my participation in the project. Besides technical excellence and great leadership, Dr. Mathis always demonstrated an extraordinary willingness to help even when faced with quite unusual requests such as proofreading a paper of mine late evening for a just-in-time submission.

This thesis required a considerable amount of implementation effort. Fortunately, I enjoyed valuable help of students on this task. Alexandre Coster implemented the first version of the dataset generation tool used in the experiments reported in Chapter 3 and 4 and Fernanda S. Pimenta implemented several of the similarity functions discussed in Chapter 3.

Thanks are also given to Manuela Burkart, Steffen Reithermann, and Lothar Gauß for administrative and technical support.

Last but not least, many thanks to all members of the Lehrgebiet Informationssysteme, external collaborators, and friends who have directly or indirectly contributed to the completion of this thesis.

This work was mostly supported by CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) under grant BEX1129/040. Further support of DFG (Deutsche Forschungsgemeinschaft) is also acknowledged. Finally, the last three months were supported by DAAD (Deutscher Akademischer Austausch Dienst).

Leonardo Andrade Ribeiro

Kaiserslautern, August 2010

Abstract

A prime motivation for using XML to directly represent pieces of information is the ability of supporting ad-hoc or "schema-later" settings. In such scenarios, modeling data under loose data constraints is essential. Of course, the flexibility of XML comes at a price: the absence of a rigid, regular, and homogeneous structure makes many aspects of data management more challenging. Such malleable data formats can also lead to severe information quality problems, because the risk of storing inconsistent and incorrect data is greatly increased. A prominent example of such problems is the appearance of the so-called *fuzzy duplicates*, i.e., multiple and non-identical representations of a real-world entity.

Similarity joins correlating XML document fragments that are similar can be used as core operators to support the identification of fuzzy duplicates. However, similarity assessment is especially difficult on XML datasets because structure, besides textual information, may exhibit variations in document fragments representing the same real-world entity. Moreover, similarity computation is substantially more expensive for tree-structured objects and, thus, is a serious performance concern.

This thesis describes the design and implementation of an effective, flexible, and high-performance XML-based similarity join framework. As main contributions, we present novel *structure-conscious* similarity functions for XML trees — either considering XML structure in isolation or combined with textual information — , mechanisms to support the selection of relevant information from XML trees and organization of this information into a suitable format for similarity calculation, and efficient algorithms for large-scale identification of similar, set-represented objects. Finally, we validate the applicability of our techniques by integrating our framework into a native XML database management system; in this context we address several issues around the integration of similarity operations into traditional database architectures.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

XML — the Extensible Markup Language — became a W3C recommendation in the beginning of 1998 with initial focus on serving as markup language for structured documents on the Web. Shortly thereafter, XML also enjoyed overwhelming popularity as common transport syntax for data exchange providing platform and data model independence for application integration. For database systems, publishing data as XML is not only important to supporting application interoperability but also to providing a hierarchical *presentation model* [JCE+07] that matches the common users' view of information, thereby alleviating the impedance mismatch problem. Moreover, XML has been widely recognized as appropriate logical data model to describe the semi-structured nature of data commonly found on the Web [Via01]. Thus, it comes as no surprise that all major commercial database systems have been extended with functionality to storing, querying, and updating XML data [BCJ+05, MLK+05, Rys05].

As XML continues its path to becoming the universal information model, large-scale XML repositories proliferate. Very often, such XML repositories are non-schematic, or have multiple, evolving, or versioned schemas. In fact, a prime motivation for using XML to directly represent pieces of information is the ability of supporting ad-hoc or "schema-later" settings. For example, the flexibility of XML can be exploited to reduce the upfront cost of data integration services, because documents originated from multiple sources can be stored without prior schema reconciliation and queried afterwards in a best-effort manner — such approach to data integration is often referred to as dataspace systems [HFM06]. As another example, features of the XML data model, such as optional elements and mixed content, allow incremental schema design on operational databases without requiring data migration; and on the application side, query integrity on evolving schemas can be preserved by using the descendant relationship to address parts of XML documents [Sed05].

Of course, the flexibility of XML comes at a price: the absence of a rigid, regular, and homogeneous structure makes many aspects of data management more challenging, such as transaction processing [Hau05], information retrieval [Dop08], query optimization [Mat09], design of efficient storage and access methods [Mat09], and statistics estimation [dAMF10]. Loose data constraints can also lead to severe data quality problems, because the risk of storing inconsistent and incorrect data is greatly increased. A prominent example of such problems is the appearance of the so-called *fuzzy duplicates*, i.e., multiple and non-identical representations of a real-world entity. Complications caused by such redundant information abound in common business practices. Some examples are misleading results of decision support queries due to erroneously inflated statistics, inability of correlating information related to the same customer, and unnecessarily repeated operations, e.g., mailing, billing, and leasing of equipment. The problem of determining whether two differently represented entities are fuzzy duplicates is commonly referred to as the *entity matching* (EM) problem and has been actively investigated by several research communities including databases, information retrieval, and machine learning.

In relational databases, fuzzy duplicates arise due to a variety of reasons, such as typographical errors and misspellings during data entry, incomplete information, and different naming conventions when multiple and independent data sources storing overlapping information are integrated. In XML databases, deviations in documents representing the same real-world entity may arise not only owing to varying content information but also because structure can diverge. For example, two documents containing approximately or exactly the same data can be arranged in quite different structures. Moreover, even documents following the same schematic rule (i.e., a DTD or XML Schema) may present structural heterogeneity owing to the presence of optional elements or attributes.

Similarity join is a fundamental operation to support EM applications. This particular kind of join operation employs a predicate of type $sf(r, s) \geq \tau$, where r and s are operand entities to be matched, sf is a *similarity function*, and τ is a constant threshold. In typical EM applications, pairs satisfying a similarity join can be classified as potential duplicates and selected for later analysis. In addition, similarity join can be employed in several other stages as well, for example, to reduce the number of candidate pairs considered by some other, usually much more expensive similarity function — an operation known as blocking — and to support training-set construction for machine learning approaches [BM03b]. Therefore, an efficient and effective similarity join implementation that can be used as core operator in numerous EM solutions is of paramount importance.

Database management systems are convenient platforms to provide support for similarity joins. Joining data based on some criterion is a primitive DBMS operation and several built-in components of off-the-shelf DBMSs, such as indexes, statistics, and physical operators, can be leveraged for the realization of similarity operators. In this vein, some work has proposed to enrich the DBMS repertoire of physical operators with basic data quality algorithms [CGK06] — similarity joins are prime examples of such algorithms. As a result, DBMSs can play a pivotal role in generic data cleaning solutions instead of merely serving as an expensive and underused file system delivering large amounts of data to external applications.

In a broader view, embedding of similarity operators into database engines lies at the core of the long-term goal of integrating database and information retrieval technologies (DB/IR integration, for short) [CRW05]. The interest in DB/IR integration has been (re)gaining strong momentum over the last few years due to a confluence of emerging and interrelated demands: *automatic ranking and keyword search in relational databases* to improve support for text data [CD09], to deal with the "too many answers" problem of SQL queries [CDHW06], and to increase database usability [JCE+07]; *structured query models for IR engines*, e.g., faceted search [YSLH03] and entity-search [CYC07]; and *managing of structured data on the Web* — such data has increasingly been accessed from static HTML tables [CHW+08] and from the Deep Web [Ber01], distilled from unstructured data by large-scale information extraction applications [Sar08], and generated by Web 2.0 technologies such as mashups and social tagging [AYMH+08]. In all these scenarios, there is the need of seamless coupling of structured and unstructured data, which is best served by an integrated DB/IR architecture [CRW05]. Finally, because the XML data model is the common choice to describe semi-structured data, XML database management systems (XDBMS, for short) are natural candidates to provide DB/IR integration.

1.1 Requirements for Similarity Joins on XML Trees

XML is commonly modeled as rooted labeled (sometimes ordered) trees, with data values associated to (part of) the leaves. Most similarity join techniques proposed so far address data of string type [KSS06]. However, XML trees are much more complex data structures than strings and so is the corresponding similarity evaluation. More specifically, we identify the following major challenges to realizing similarity joins on schemaless XML databases effectively and efficiently:

- *Structure-conscious similarity evaluation:* The similarity function used as similarity join predicate should address the hierarchical structure of the documents. Even assuming that element labels are drawn from a common vocabulary — a typical scenario for real-world (domain-specific) heterogeneous XML stores, where usually there is some understanding of a set of interesting labels [LM09] — or mapped onto an ontological space in a pre-processing step [TSW03, MMR05], it is still challenging to assess the similarity of arbitrarily arranged structures. For example, among documents in a dataset, element nodes with the same tag label can appear at different contexts (e.g., hierarchical nesting, sibling ordering, and containing varying number of descendant nodes) and with different frequencies. In EM applications, similarity evaluation between two strings typically aim at identifying when one string is an erroneous variant of the other (e.g., due to misspelling) [Nav01]. For XML trees, a structure-conscious similarity function needs to capture when two XML documents are *modeling variants* of each other; such abstract notion of similarity is much more difficult to define and to measure.

- *Combination of structural and textual similarity:* Many approaches focus on the structural similarity only [ABG05, ABDG08, But04, FMM+05, Hel07, NJ02]; text nodes are either stripped away, before the matching process is initiated, or only simple equality operations on text values are considered. While structural information certainly conveys useful semantic information, most of the discriminating power of real-life XML data, i.e., the items of identifying information, which allow to distinguish documents in a collection from each another (e.g., keys), is assumed to be represented by text nodes. As a result, the resulting similarity notion is often too “loose” for EM applications or ineffective for databases exhibiting poor textual quality. Considering textual and structural similarity together brings up the issue of combining similarity evaluation results, which can be seen as an instance of the *combination of evidence* problem [Cro00]. While combination of evidence is necessary for structured data if multiple fields are used for similarity calculation [FS69], the duality of structure and text makes it an inherent component when assessing the similarity of XML trees. Combining structural and textual similarity is not trivial, however. The common solution of adopting a linear combination of similarity function values can be problematic. Besides having different semantics, text and structural patterns usually present very variable frequency distributions across different databases; optimal weights for the corresponding similarity results are likely to vary accordingly. As a result, there is a strong dependence on comprehensive training data (for supervised learning approaches) or intensive user guidance, both of them are not available in many real-world scenarios.

- *Entity description selection:* Frequently, only part of the available information about an entity is interesting for similarity matching. For example, while `author` and `title` are very significant in a database about publications, `year` is expected to be much less useful. Such interesting pieces of information constitute the *entity description*, which is exploited to identify fuzzy duplicates. For structured data, entity description elements are usually gathered and placed in a single location (e.g., table in relational databases), before the EM process begins. In heterogeneous XML databases, however, such elements may be dispersed in different parts of the XML trees owing to the structural heterogeneity. Using schema-aware query languages like XQuery [BCF+07] or XPath [BBC+07] to collect all necessary entity information as proposed in [WN05] can be impractical, because the user may have only limited knowledge about the underlying structure. Even if the structure is known, it would be clumsy and error-prone to specify multiple queries using strict structural constraints, because the unified schema of heterogeneous XML databases is often very complex. A better approach is to adopt a flexible query model based on XML search [AYL06]: structural constraints are interpreted *approximately*, i.e., the structural context of the entity descriptions do not need to match exactly the structural constraints expressed in the query. Complementary to the flexible query model, it is important to provide the user with interactive exploration capabilities to support the selection of interesting entity descriptions. Further issues are coupling of flexible location of entity descriptions with the similarity join processing for performance reasons, and the definition of a clean semantics for delimitation between structural and textual entity descriptions.

- *Efficiency:* Several measures for assessing the similarity between tree-structured data are computationally expensive. One example of such a measure is the widely used *tree edit distance* (TED), which is defined as the minimal sequence of edit operations (node insertion, node deletion, and node relabeling) that transforms one tree into another [Tai79]. The best known algorithm for computing the tree edit distance for ordered trees has worst-case running time of $O(n^3)$ [DMRW07], where n is the number of nodes; for unordered trees, the problem has been shown to be NP-complete [ZSS92]. Even worse, some measures do not easily lend themselves to effective minimization of the number of pairwise comparisons, thereby requiring similarity evaluation between every pair of input XML trees. Obviously, these limitations exclude such measures from being used in similarity operations on large databases. Moreover, other requirements mentioned previously, namely combination of structural and textual similarity and coupling of approximate locations of entity descriptions with query processing, substantially complicate the design of efficient and scalable algorithms.

In addition to XML-motivated requirements, versatility in supporting a variety of notions of similarity is an orthogonal concern in the context of EM applications. It is well known that no single similarity function is the best for all applications and scenarios (e.g., see [Cohen et al. 2003] for string similarity functions). Hence, it is very desirable to have, instead of a single measure, a broader class of similarity functions, in which different notions of similarity can be easily obtained by changing simple parameters. Finally, regarding the integration of similarity joins in a DBMS environment, an important challenge is to exploit DBMS internals (e.g., physical operators, indexes) without negatively interfering with the operation of other components of the system or sacrificing the accuracy of the similarity evaluations.

1.2 Thesis Contributions

In this thesis, we present an XML-based similarity join framework addressing all the issues outlined above. As key design decision, we focus on the class of token-based similarity functions (aka set-overlap-based similarity functions). This class of functions ascertains the similarity between two entities of interest, i.e., XML trees, by measuring the overlap between *token sets* generated from these entities. This approach has three main advantages. First, we can measure textual and structural similarity between XML trees, jointly or in isolation, by operating on token sets representing text, structure, or both, in a unified framework. Second, we can obtain a very rich similarity space by varying the methods for generating token sets, associating weights to set elements, measuring the set overlap, or any combination thereof. Third, token-based similarity functions are inexpensive to calculate, and it is possible to leverage a wealth of techniques for set similarity joins on strings (e.g., [SK04, CGK06]). In this context, the contributions of this thesis are summarized as follows:

- We present a new method to generate structural token sets for *unordered* XML trees. Our method exploits *path synopses* — fundamental structures for efficient XPath query evaluation and therefore ubiquitously supported by XDBMSs — to produce compact and high-quality structural representations of XML documents. We experimentally compare our approach against other token-based similarity functions as well as approaches based on edit distance, entropy, and Discrete Fourier Transform on several different datasets. Our approach provides accurate results in all the settings investigated and outperform previous state-of-the-art techniques. Moreover, it delivers much smaller token sets — bounded by the number of paths in a tree — , the tokens lend themselves to very compact representations and can be generated for free using simple data structures. In this connection, we investigate different weighting strategies for structural tokens.

- We study strategies for combining structural and textual similarity. We consider combination approaches at the similarity score-level and token-level. For unordered trees, we use structural token sets based on path summaries together with textual token sets based on *q-grams* [Ukk92]. For ordered trees, we propose an extension to the concept of *pq*-grams [ABG05] to capture textual information, the so-called *extended pq-grams*. In this context, we also study ways of supporting the entity description selection. In our similarity join framework, the entity description consist of XML structure and user-selected string values; structural and textual token sets are then generated from them. We propose a mechanism to support the selection of string values from XML trees based on a compact structure that approximately subsumes all paths contained in a path synopsis. This structure is represented by short memory-resident inverted lists and supports approximate path matching and user interaction.

- We present a new index-based algorithm for set similarity joins. Following an approach completely different from previous work, we focus on the reduction of the computational cost for candidate generation as opposed to lowering the number of candidates. To this end, we introduce the concept of *min-prefix*, a generalization of the *prefix filtering* concept [SK04, CGK06], which allows to *dynamically* and *safely* minimize the length of the inverted lists; hence, a larger number of irrelevant candidate pairs are never considered and, in turn, a drastic decrease of the candidate generation time is achieved. As a side-effect, the workload of the verification phase is increased. We optimize this phase by stopping as early as possible the computation of candidate pairs that do not meet the overlap constraint. We also improve the overlap score accumulation by storing scores and auxiliary information within the indexed set itself instead of using a hash-based map. Finally, we consider disk-based and parallel versions of the algorithm. Our experimental findings confirm that our algorithm consistently outperforms previous ones under several different data distributions and configuration parameters, for both unweighted and weighted sets.

- We address the integration of our framework into an XDBMS. We focus on the system embedding of similarity operators on top of the storage layer by using index-based location of qualified XML fragments, leveraging of physical algebra for XQuery processing, and enhancing performance using pipelined evaluation. The resulting operators can be combined (e.g., by delivering the resulting nodes in document order) with regular XML queries to compose processing logic which enables more complex data integration solutions. Furthermore, we exploit XDBMS's storage model and node identification mechanism to efficiently generate XML tree representations for similarity calculation. We experimentally demonstrate that our approach provides scalability and achieves seamless integration with other system components.

1.3 Thesis Outline

Below we give a summary of the upcoming chapters in the thesis.

In Chapter 2, we provide background material for following discussions on the subject area. We first describe the main aspects of the EM problem in detail. We use a conceptual framework based on early EM work as a common ground for our discussion. Recent EM approaches representing a more marked departure from our reference framework are discussed as well. We highlight the importance of similarity joins as a foundational operator by providing several examples drawn from several different EM solutions. We then turn our focus to XML data. We first introduce the XML data model and make underlying assumptions explicit, before we formally define XML similarity joins. Next, we discuss two popular classes of similarity functions, namely edit-distance and token-based similarity functions. Finally, we describe our strategy for measuring the effectiveness of similarity join algorithms. Concepts presented in this chapter are of interest in all upcoming chapters; using these concepts, further background concepts, whose scope is constricted in each case to a specific aspect of this thesis, are described in the respective chapters.

In Chapter 3, we introduce our novel approach for measuring the structural similarity of XML trees and present a thorough study comparing our approach against a highly representative group of measures from previous work. We describe the existing approaches in Section 3.1. In Section 3.2, we introduce the concept of *Path Cluster Identifiers* (PCIs), describe its generation process, and present the corresponding similarity function. We then conduct extensive experiments on several different datasets; besides the comparison between several structural similarity measures, we also explore different weighting strategies for structural tokens.

Chapter 4 is dedicated to the combination of textual and structural similarity and entity description selection. We present combination strategies for ordered and unordered trees. For the former, we introduce the concept of *extended pq-grams*, which extends *pq*-grams tokens [ABG10] with textual information. For this concept, we propose three versions of token generation methods and provide their theoretical analysis. For the latter, we present two strategies for combining *PCI*-based structural description with textual information. Next, we describe the Path Cluster Summary (PCS), a compact structure used to select pieces of textual information that will compose XML entity descriptions. Extensive experiments comparing the combination models proposed are then presented and interpreted.

After having converted XML trees into sets of tokens, the remaining processing consists of identifying those sets sharing enough tokens, i.e., a set similarity join has to be performed. Chapter 5 presents a new algorithm for this task. Main optimization techniques from previous work are outlined, before we present the key insights leading to the min-prefix concept and its corresponding index-based algorithm. Further optimizations and the case, where tokens have associated weights, are addressed, before we present experiments measuring the runtime performance and scalability of our algorithms and comparing them against previous, state-of-the-art set similarity join algorithms; in this context, we also identify the most important characteristics of the input data driving the performance of the set similarity joins algorithms under study.

In Chapter 6, we put it all together into a prototype XML DBMS called XTC (XML Transactional Coordinator). We first sketch the architectural design of XTC with emphasis on the storage model and access path mechanisms. We then describe scan and token generation operators highlighting the exploitation of database internals to improve efficiency. We also address updatability of auxiliary structures. Further, our integration of similarity operators with the existing physical algebra to produce pipelined query execution plans is presented. We discuss further steps for providing tighter integration of our framework into XTC before we provide runtime performance experiments and scalability studies.

Finally, in Chapter 7, we wrap up this thesis with the conclusions and outline interesting future work.

Chapter 2

Background

There is a large class of database application domains where the traditional paradigm of Boolean queries based on *exact matching* of precisely defined and represented data is insufficient or inadequate. Just a few examples of such application domains are video and image retrieval [FSN+95], bioinformatics [Coh04], information extraction [Sar08], and dataspace systems [HFM06]; all these examples have to deal with complex, heterogeneous, unstructured, and imprecise data. Equality comparisons on such data are meaningless. Moreover, exact matching is sometimes ruled out by the inherent purpose of the task at hand. For instance, the goal in (ranked) text retrieval is to find documents that are relevant to some information needs and a document may be deemed as relevant, even though it contains none of the query terms; in clustering analysis, the aim is to group objects that are similar. In all these cases, it is appropriate to adopt the more general paradigm of *similarity matching* where objects are matched in terms of their similarity. The notion of similarity is typically quantified by a *similarity function*, sometimes with support of external information such as statistics and tables of user-specified transformations. In this context, *similarity join* is the counterpart of the relational join operator for combining information. While several semantics are conceivable, the most studied type of similarity join pairs objects from a database, whose similarity according to a similarity function is not less than a specified threshold.

In this thesis, we focus on similarity joins in the context of the *EM* problem. The basic definition of this problem precludes exact matching: the goal in EM is to identify *non-identical* representations of an entity, i.e., *fuzzy duplicates* (or simply duplicates, whenever clear from the context). At a higher level of abstraction, the EM problem can be viewed as similarity joins employing a similarity function that mathematically approximates the notion of duplication. In practice, however, the concept of duplication can hardly be captured in a closed-form formula or by simple algorithms. Instead, identification of duplicates is normally conducted in a very complex process involving several stages and requiring human supervision. Nevertheless, similarity joins can play different roles in an EM process as will be illustrated later. In this connection, similarity functions are of paramount importance. Even though similarity alone is often insufficient to safely classify two entities as duplicates, a basic premise is that "good" similarity functions for EM return higher values for pairs of duplicates than for pairs of non-duplicates. We follow this premise in this thesis.[1]

Here, because the entities of interest are XML trees, characteristics of the XML data model have to guide the design of suitable similarity functions. Of course, given the large number of similarity calculations involved during similarity join processing, efficiency is also a crucial requirement. In this vein, although we do not assume that all XML data complies with the same schema, it would be infeasible to develop efficiently computable functions or algorithms on arbitrarily arranged XML documents. Thus, some assumptions about the input datasets are necessary to reasonably evaluate similarity joins on XML data.

The goal of this chapter is to pave the way for our study on XML similarity joins. In Section 2.1, we start by providing background material on EM systems and emphasizing the use of similarity joins as a foundational operator in several settings. We then turn our focus to XML data. In Section 2.2, we define our XML data model and make the underlying assumptions explicit, before we formally state the XML similarity join operation; at the end of the section, we discuss some related approaches. In Section 2.3, we describe two important classes of similarity functions and analyze them under criteria pertinent to the context of our work. The strategy adopted in this thesis for measuring the effectiveness of similarity functions is presented in Section 2.4. Along the discussion of these sections, we will introduce some notation and provide references to relevant work. Finally, we conclude this chapter in Section 2.5.

2.1 Entity Matching

It is reasonable to conjecture that the concern about duplicates is coexistent with the practice of recording information itself. As already mentioned in Chapter 1, there are many reasons for the appearance of such duplicates in a dataset. For instance, consider the sample data from a hypothetical relational database shown in Figure 2.1. It is apparent that all the three tuples refer to the same entity, yet they present syntactical differences due to several reasons: misspellings ("Leonardo Hibeiro" and "Kasesrlatern"); different naming conventions such as use of first and middle initial in the name attribute ("L. A. Ribeiro"); abbreviations ("KL"); incomplete information ("Gottlieb Str."); typing errors ("111-1322" instead of "111-2322"); and spelling variations ("Gottlieb-Daimler-Straße" vs. "Gottlieb Daimler Strasse").

EM is probably the most important data quality activity. The presence of duplicates in databases violates basic data integrity principles and degrades the quality of the data delivered to application programs. This fact can lead to a myriad of problems in all sorts of Information Systems (IS), e.g., poor service quality and application program malfunction, very often causing severe economic losses [III09].

[1]Later, we will see that the above premise is strongly related to the *monotonicity property*, which has been exploited in prior work on EM.

	Name	City	Phno	Address
r_1	Leonardo Ribeiro	KL	111-2322	Gottlieb-Daimler-Straße
r_2	Leonardo Hibeiro	Kaiserslautern	111-1322	Gottlieb Str.
r_3	L. A. Ribeiro	Kaserslatern	111-2322	Gottlieb Daimler Strasse

Figure 2.1: Example of fuzzy duplicates

Entity matching is of particular importance for data integration systems. For example, in the *data warehousing* approach, operational data sources distributed across an enterprise are consolidated in a single database. Entities represented in multiple data sources will inevitably be duplicated in the data warehouse thereby jeopardizing the results of decision-support queries. Thus, EM is an essential component of the repertoire of Extract-Transform-Load (ETL) tools. In the *virtual data integration* approach, the integrated data is not materialized; in contrast, a virtual schema is constructed from the data sources. Query requests posed over the virtual schema are translated into queries on the data sources and the results are integrated before being returned to the user. In this scenario, duplicated data in the query result have to be dealt on-the-fly, which makes EM much more challenging. The implementation of this approach is often called Enterprise Information Integration (EII).[2] Furthermore, many data integration systems heavily rely on the ability of identifying the same entity across different organizations to deliver their functionality. For instance, consider a Healthcare Network (HN) connecting various IS of healthcare institutions [Len05]. A major benefit of HN systems is the ability to access an individual's lifetime health and medical information, which is scattered along each connected system. To this end, the capability of correlating information of the same patient across different systems is essential. Finally, in the Web search context, EM has been recognized as a fundamental operation to enable the next generation of search engines [DKP+09].

Not surprisingly, the EM problem has a long history of investigation conducted by several research communities spanning databases, information retrieval, machine learning, and statistics, frequently under different names, including record linkage, entity identification, deduplication, and near-duplicate identification [Win06, KSS06, EIV07]. As a result, a vast body of work is available addressing many aspects of this topic from different perspectives. In the following, we survey results of these research activities. We will not be exhaustive, however; a comprehensive survey of the voluminous literature on EM would far exceed the scope of this thesis. We refer the reader to the recent surveys [Win06, EIV07], which cover many material missed here. Nevertheless, this section is still relatively long. Our aim is to provide a representative, albeit incomplete, overview of the EM problem, describe key components that constitute a general EM framework, and portray the context in which our work

[2]Some authors use the term *data integration* to generally refer to approaches based on a virtual schema (e.g., see [Kol05]). However, we consider the term virtual data integration more intuitive.

on similarity joins is situated. In Section 2.1.1, we begin with seminal work on the EM problem, which introduced several fundamental ideas. The main components of an EM system are detailed in Section 2.1.2. Recent trends are discussed in Section 2.1.3. Finally, we illustrate the application of similarity joins to support several EM strategies in Section 2.1.4.

2.1.1 Early Work

Newcombe et al. [NKAJ59] pioneered the computerized identification of fuzzy duplicates in the late 1950s. The studies were conducted in the context of epidemiological analysis and the task — termed as *record linkage* — consisted of matching birth records to marriage records. Several crucial insights were presented in this work:

- Exact matching on comparison fields, such as surnames, is problematic owing to spelling discrepancies. The authors proposed phonetic-based similarity matching to deal with this problem. Using the *Soundex* algorithm [Rus18], they were able to identify 2/3 of the spelling variations in family names.

- It is unfeasible to consider each element from the cross-product of two (large) input files. To reduce the number of comparisons, Newcombe et al. arranged the input files according to a key derived from the Soundex code of selected fields. The aim was to bring together potential duplicates before applying the main comparison procedure. The authors also discussed how to improve the completeness of the operation (i.e., identifying more duplicates) by performing multiple independent arrangements of the data, each one based on a different subset of the comparison fields. This general idea of reducing the comparison space is currently referred to as *blocking*.

- The importance of a field value to the belief that two records are duplicates or not can be measured using the number of occurrences of this field value in the record collection. The underlying intuition is that rare values contribute more to the probability of making a correct classification than frequent values and, therefore, should receive more weight. To this end, the authors expressed the weight of a field value v, denoted by $wt(v)$, using the following log-odd ratio:

  $$wt(v) = \log_2(\mathit{freq}(D, v)) - \log_2(\mathit{freq}(N, v)) \; , \tag{2.1}$$

  where $\mathit{freq}(v, D)$ and $\mathit{freq}(v, N)$ are the frequency of v among the set of duplicates D, and the set of non-duplicates N, respectively. Because such statistics are not available — D and N are, of course, not known beforehand — the following approximation was proposed:

  $$wt(v) = \log_2(\mathit{freq}(v, C)) - \log_2(\mathit{freq}(v, C))^2 = -\log_2(\mathit{freq}(v, C)) \; , \tag{2.2}$$

where $\log_2(\mathit{freq}(v, C))$ is the frequency of v in the whole collection of records C.[3]

- The result of the comparison of different fields can be combined to obtain the *overall matching score* of a record pair; further, a *decision rule* can be employed to assign the candidate pair to D, N, or H, where H is the set of candidates requiring human inspection. The EM task can, therefore, be interpreted as a classification task. For example, the decision rule of an EM classification model can expressed as:

$$\langle r_1, r_2\rangle \in \begin{cases} D & \text{if} \quad \tau_u \leq ms(r_1, r_2) \\ H & \text{if} \quad \tau_l < ms(r_1, r_2) < \tau_u \\ N & \text{if} \quad \tau_l \geq ms(r_1, r_2) \end{cases} \tag{2.3}$$

  where $ms(r_1, r_2)$ is the overall matching score between r_1 and r_2, and τ_u and τ_l are threshold values, where $\tau_u \geq \tau_l$. Newcombe et al. used $\tau_u = 10$ and $\tau_l = -10$ in their experimental study, i.e., a record pair is selected for human inspection when the odds favoring a decision is less than 1000 to 1.

Ten years later, Fellegi and Sunter [FS69] provided a rigorous mathematical foundation to the ideas of Newcombe et al. under the Bayesian decision theory. Briefly, record pairs are represented as a comparison vector γ, which encodes the result of each field comparison (e.g., "soundex code agreement on first name" and "missing birth place in first record"). The overall matching score is given by the following likelihood ratio:

$$ms(r_1, r_2) = \frac{p(\gamma|D)}{p(\gamma|N)} \tag{2.4}$$

where $p(\gamma|D)$ and $p(\gamma|N)$ are the likelihood that γ will be observed for duplicates and non-duplicates, respectively. With the values of τ_l and τ_u based on desired error levels, it was shown that the resulting decision rule is optimal i.e., the decision rule minimizes the probability of failing to make an automatic decision (by assigning a record pair to H). Fellegi and Sunter also addressed several practical aspects: the authors assumed conditional independence of different comparison results to simplify the computation — an approach also followed by Newcombe et al. — and provided methods to estimate the conditional probabilities used in Equation (2.4) and the thresholds τ_l and τ_u; moreover, blocking methods were proposed to reduce the comparison space.

[3]The *Inverse Document Frequency* (IDF) weighting scheme, which was proposed by Spärk Jones [Jon72] more than 20 year later and became extremely popular in the Information Retrieval (IR) area, bears strong resemblance to the frequency-based approximation of Newcombe et al. in Equation (2.2). IDF as well as other weighting schemes are described in Section 2.3.2.

2.1.2 Conceptual EM Framework

Newcombe et al. and Fellegi and Sunter laid the groundwork for large parts of the subsequent research on EM. In fact, blocking, similarity matching, and classification based on the combination of comparison results are prevalent in EM approaches. In addition, evaluation is typically preceded by an off-line pre-processing step and followed by a clustering algorithm applied on the result of the classification model. Together, these steps form a general EM framework which we refer henceforth to as the *Newcombe et al. and Fellegi and Sunter* (*NFS*) *framework*.

Figure 2.2 shows how the NFS framework is structured into its main components. In the pre-processing step, the *Classification Model Design* defines several parameters necessary for EM evaluation (e.g., comparison fields and similarity functions). Such parameters can be manually specified by a domain expert or obtained from learning methods (using labeled examples). In the evaluation phase, the *Blocking* component generates a set of candidate pairs whose size is (expected to be) substantially smaller than the cross-product between the input datasets; alternatively, Blocking can be applied on a single dataset as a self-join. Following the evaluation course, *Matching* compares the selected fields of each candidate pair using their corresponding similarity functions to produce a set of comparison vectors. The output of Matching is then delivered to the *Classification* component, which designates candidate pairs to D, N, or H. *Data Transformation* encompasses any transformation that makes the data amenable for duplicate identification, ranging from dimension reduction to schema mapping methods. Note that Data Transformation spans pre-processing and evaluation, i.e., it can take place on either phase depending on the strategy adopted. Finally, as a post-processing step, *Clustering* uses the result of Classification — represented as a *duplicate graph* — to group all entities that refer to the same entity, possibly correcting matching inconsistencies.

Although closely followed by many EM systems [EEV02, WN05], the NFS framework is purely conceptual and digressions from the structure suggested in Figure 2.2 are common in practice. For instance, Matching and Classification are often integrated into a single operation and the set N is not materialized. Nevertheless, the NFS framework serves well as reference architecture for our discussion. In Section 2.1.3, we cover recent EM techniques that represent a more marked departure from the NFS framework and discuss adjustments on the framework to accomodate them.

In the rest of this section, we will continue using the terms *field* and *record* for referring to entity representations. We shall change our terminology from Section 2.2 on, as we focus on XML data.

Classification Model Design

In this section, we discuss main aspects of the design of accurate classification models. This activity is conducted off-line in the pre-processing phase and may well be the most time-consuming stage of an EM process. The design space is essentially constituted by the selection of comparison fields and similarity functions, specification

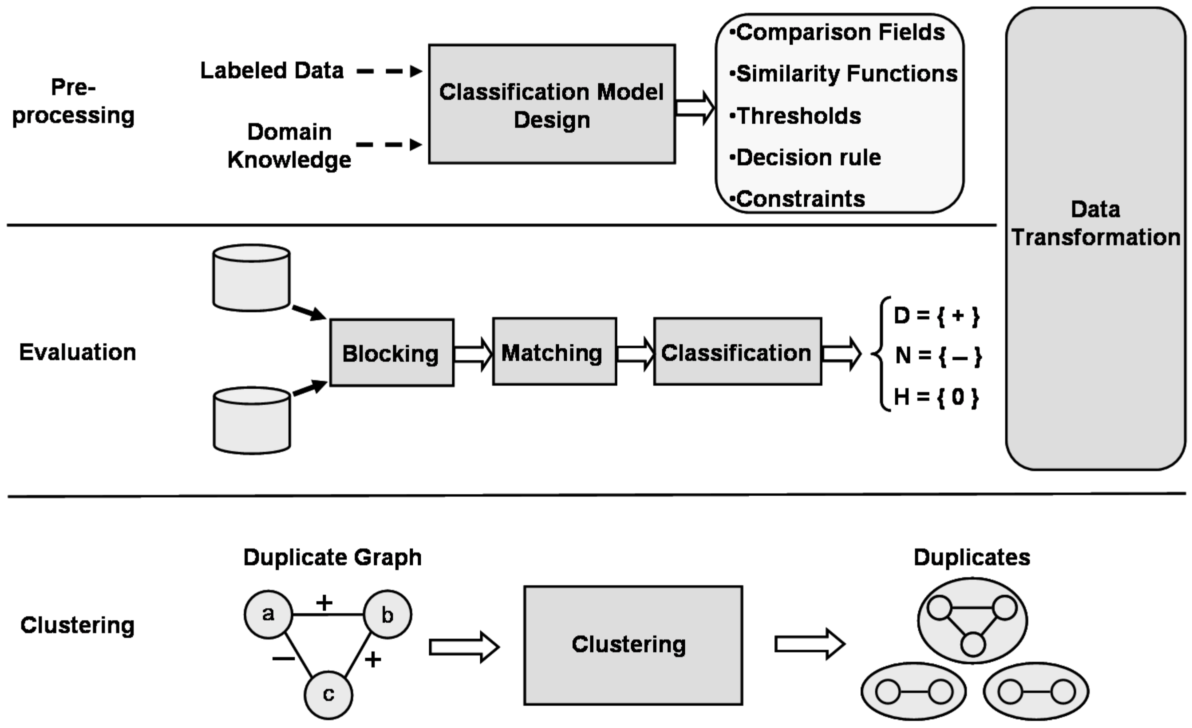


Figure 2.2: The NFS framework

of thresholds, and the method for combining all these elements into a decision rule.

Some approaches exploit constraints to improve effectiveness. Example of such constraints are: "one entity can be matched with at most another entity", "the number of publications must be the same for author", and "two specific entities cannot match". See [CSGK07] for a taxonomy of clustering constraints in the context of EM. We do not cover the selection of constraints here because its interest is limited to specific approaches; we revisit constraint-based methods in Section 2.1.3, though.

A variety of data exploration and data preparation tasks may be needed to support classification model design. We discuss several data transformation operations in the next section; however, a comprehensive review on data exploration and data preparation is far beyond the scope of this thesis. We refer the reader to [Pyl99] for a textbook on this topic.

Comparison Fields An ideal set of comparison fields would characterize entities in such a way that values yielded by similarity functions would be very high for duplicates and very low for non-duplicates, thereby making the classification model trivial. Domain knowledge provided by an expert is essential; even learning models which find a set comparison fields from labeled examples (e.g., [CCGK07]), still relies on expert guidance to obtain a good initial set of fields. Moreover, the selection of comparison fields, either automatic or user-defined, has to be made in conjunction with the corresponding similarity function and data transformation operation. Finally, it is possible to create new fields by concatenating multiple fields — note that, for some similarity functions, the order of the concatenated fields is relevant — or, conversely, by text segmentation (we discuss text segmentation shortly).

Several criteria can be observed for choosing appropriate comparison fields: discriminating power, underlying data quality and low-probability of misreporting (e.g., fields whose consistency is enforced by integrity constraints), availability of external resources to support similarity assessment (e.g., table of abbreviations), and performance (e.g., favoring of fields containing shorter strings).

Another concern is the relationship between comparison fields. Newcombe et al. [NKAJ59] obtained better separation between duplicates and non-duplicates by using additional fields. Concretely, the number of records falling in the uncertain region (i.e., the records designated to H) was reduced by a factor of 3 after the inclusion of age information to the set of comparison fields. Additional fields do not always mean better matching, however. The main issue is the lack of independence between comparison fields. Several approaches (e.g., [NKAJ59] and [FS69]) assume that comparison fields are conditionally independent distributed to make the calculation of the overall matching score computationally faster; for such approaches, the addition of correlated fields lead to overestimation of the matching score, which may negatively affect accuracy. While it is possible to use models that account for dependencies — for example, Winkler [Win88] proposed the *Expectation-Maximization* algorithm to estimate the likelihoods without requiring conditional independence — , models as such are invariably computationally more expensive because the complexity of the inference problem is increased. Besides, it is intuitive to assume, and theoretically verifiable using Bayesian formalism [Pea88], that the impact of new evidence to the belief in a hypothesis is inversely proportional to its degree of dependency on previous evidence. Putting it in the EM context, the contribution of a field to the classification of a pair of entities as matching or non-matching is reduced if this field is correlated with other comparison fields. In summary, independence between comparison fields can affect both performance and accuracy of EM and therefore is an important criterion for the design of classification models.

Besides the target entity, comparison fields can be selected from sources of related information. For example, in bibliographic datasets, information about papers can be used to support matching of authors. Such approach defines the notion of similarity based on "co-occurrence" between entities [ACG02]: in the previous example, two authors will have high similarity if they share a large number of papers. The notion of co-occurrence similarity is inherent when matching hierarchically represented entities, e.g., dimensional hierarchies in datawarehouses or XML data. We discuss this aspect further in Section 2.2.

Similarity Functions The concept of similarity is intrinsically dependent on the application domain. Thus, no single similarity function is overall the best across all scenarios [CRF03, SB02, CGK06]. For example, a similarity function which accurately captures misspellings and other character-level variations (e.g., *Ribeiro* and *Hibeiro*) is likely to perform poorly on fields characterized by the presence of abbreviations (e.g., *Kaiserslatuern* and *KL*). This observation strengthens the importance of jointly selecting comparison fields and similarity functions. Moreover, it makes a strong case for versatile classes of similarity functions in which different notions of similarity can be easily obtained by simple reformulations or changing of parameters. In this connection, besides selection, also tuning of similarity functions to specific data domains can be carried out in the design phase. For example, Bilenko et al. [BM03a] described how to learn an edit distance model with affine gaps and a weighting scheme for similarity computation based on vector spaces.

Thresholds Most EM approaches employ similarity thresholds to exclude candidate pairs that are unlikely to be duplicates and ultimately yield the final matching status of a candidate pair. Thresholds can be user-specified or learned and used in similarity predicates involving fields or whole entities.

Using thresholds poses a trade-off with respects to the exactness and completeness of results returned by the similarity predicate. For example, raising the threshold normally implies in results with higher fraction of duplicates (true matches), i.e., increased *precision*; however, more duplicates may be missed, i.e., *recall* is decreased.[4] Conversely, lowering the threshold may add more duplicates to the result and increase the recall, but, at the expense of adding non-duplicates too, thereby hurting precision. On the one hand, recall is commonly favored over precision because non-duplicates in the result set can always be identified in the post-processing step whereas the only way to recover missed duplicates is re-running the EM algorithm [GIKS03]. On the other hand, some operations, in particular similarity joins, run much more faster at higher thresholds.

Besides the trade-off between precision and recall, another difficulty in specifying thresholds comes from the fact that similarity functions often return very different similarity score distributions. As a result, thresholds have to be adjusted for each similarity function. Learning methods can be employed to find an appropriated threshold configuration. In a different approach, Dorneles et al. [DNH$^+$09] used examples to create mapping of precision values to similarity scores, which allows the user to express similarity queries in terms of a precision threshold. Finally, Bhattacharya and Getoor [BG06] avoid using threshold altogether. The authors use a *Latent Dirichlet Allocation* [BNJ03] model to automatically identify the most likely number of entities in a dataset.

Decision Rules The classification of a candidate pair is performed using a decision rule. Here, we loosely treat the combination of results of each field comparison as

[4]We formally define precision and recall in Section 2.4.

a component of the decision rule as in Equation (2.3), although it can be designed and evaluated independently. Decision rules can be expressed in many ways: set of IF-THEN-ELSE clauses [WM89], sophisticated declarative rule languages [HS98], first-order logic sentences [SD06], propositional expressions in disjunctive normal form [CNS04, CCGK07], simple predicates involving a linear combination [BM03a], among others. The formulation of decision rules directly affects efficiency of EM tasks, because some formulations provide superior optimization opportunities for the Matching and Classification components as we will discuss later.

Rule-based approaches employ hand-coded rules crafted by a domain expert [WM89, HS98]. Such approaches are typically tailor-made to very specific applications usually deliver very accurate results, because many intricacies of the underlying domain are manually captured in the set of rules, and high-performance EM systems, because they lend themselves to customized implementations. A major drawback of rule-based approaches is the huge effort necessary to devise the set of rules; moreover, these approaches are too brittle for open or dynamic domains.

Learning approaches have been widely used to automatically construct decision rules. A learning algorithm receives as input a set of training data containing examples of duplicates and non-duplicates and a (possibly redundant) set of similarity functions; the output is a decision rule specifying the best way of combining and thresholding the comparison results as well as providing the final classification result. Popular learners used in previous work are Decision trees [TKM02] (D-trees), naive Bayes [SB02], and Support Vector Machines [BM03a] (SVM). Recently, Chaudhuri et al. [CCGK07] modeled the classification problem as that of identifying hyper-rectangles in a similarity space, where each dimension corresponds to an association between a comparison field and a similarity function. After embedding the record pairs of the training set into the similarity space, the problem consists of finding a set of hyper-retangles covering the maximum number of the duplicates and not more than a specified number of non-duplicates. This problem was shown to be NP-hard and the authors provided an approximate greedy algorithm based on a recursive divide-and-conquer strategy.

An issue with learning approaches is the difficulty of constructing a representative training set capturing all the subtle patterns that characterize duplicates and non-duplicates. Moreover, the number of required labeled examples tends to be large. To tackle this problem, previous work proposed active learning techniques [SB02, TKM02]. The general approach is to first train a *committee* of classifiers using a limited set of examples and then interactively increment the training set with examples that, after being labeled by the user, will provide the highest information gain to the learning process. Such informative examples are those about which the committee is most uncertain. A simple way to assess the "degree of uncertainty" of an example is observing the disagreement of the committee on its classification: uncertain examples lead to greater disagreement among the classifiers. Using an active learning strategy, the number of examples needed for the learner to achieve peak accuracy can be reduced by orders of magnitude.

Further Combination Approaches A closely related problem is the *combination of evidence* in IR [Cro00], where results from different data sources or retrieval strategies are combined to improve the quality of the results. Different "pieces of evidence" for a retrieval process can be obtained by using multiple query representations (e.g., representations derived from relevance feedback and query expansion techniques), document representations (e.g., using different fields of document's internal structure such as title, in-link, meta tag keywords, and elements of XML documents), retrieval models (vector space model, probabilistic model, etc), and ranking algorithms (e.g., different weighting schemes and similarity functions); also, evidence can refer to results from independent search engines[5]. Drawing a parallel between combination of evidence in IR and EM is straightforward. Retrieval systems and EM systems can both be viewed as classifiers: the first assigns documents to the classes *relevant* and *non-relevant*, while the second assigns entities to the classes duplicates and non-duplicates (and additionally to the class *possible duplicate*, i.e., to the set H). Hence, combination of evidence in IR and EM can be modeled as a combination of classifiers [6] and many techniques and studies on EM and combination of evidence are applicable to one another.

Lee [Lee97] analyzed the output of multiple classifiers and concluded that the best combinations were between classifiers that retrieve similar sets of relevant documents and dissimilar sets of non-relevant documents. In this context, simple combination strategies such as CombSUM and CombMNZ, which are based on the summation of the similarity scores, have been shown to be effective because they favor documents retrieved by more classifiers. The results of these approaches can be further improved by performing a linear normalization of scores to have all values in the interval $[0, 1]$. Several approaches adopt the linear combination of scores — including variants of CombSUM and CombMNZ — where the weights are either hand-tuned or learned. These techniques are similar to the combining approaches employed in EM.

Some approaches exploit the induced ranking to perform combination in lieu of the similarity values themselves [FKS03, GKMS04]. Common combination functions are adaptations of standard methods for comparing permutations, such as *Spearman's footrule* and *Kendall's tau* distances (see [FKS03], and references therein). However, some studies have shown that ranking-based combination is generally outperformed by score-based combination [Lee97]. More importantly, while the main goal of combination of evidence in most IR settings is to produce an ordinal ranking, i.e, the score distribution is unimportant, accurate score combination is crucial in EM systems because a combined similarity value is needed to produce the final matching decision. The work in [GKMS04] presented a generalization the Spearman's footrule distance that regards the underlying score values; results of similar nature were presented in [BP09] for the Kendall's tau distance.

Finally, combination of evidence can be performed at term level. Ogilvie and Callan [OC03] inferred separate language models [PC98] for each document representation;

[5]The combination problem in this case is commonly referred to as the *meta-search problem*.

[6]This modeling for combination of evidence was first presented in [Cro00].

the probability of producing a term for a given document is calculated by a linear combination of its corresponding language models. Robertson et al. [RZT04] identified several issues with approaches that both employ weighting schemes based on document and collection statistics (term frequency, document frequency, document length, etc) and further adopt a linear combination of the scores obtained from every document representation. For example, using the Okapi BM25 weighting scheme [RW94], a linear combination of scores would excessively amplify the weight of term appearing in several representations of a document. To avoid this and other problems, the authors proposed a linear combination of term frequencies, which is performed before the calculation of the term weights. In the XML context, Carmel et al. [CMM$^+$03] weighted terms with respects to their *context*, where the context of a term is given by the *path* leading to it from the root node of the XML document. Statistics used by the weighting function are constrained to the context of a given a term and all the other contexts that are *similar* to the term context. In the EM scenario, term-level combination can be employed on groups of comparison fields that are associated to the same single similarity function. In particular, the approach of [CMM$^+$03] is close in spirit to our own approach to combinining structural and textual similarity, which is introduced in Chapter 4; also in this chapter, we will present techniques for generating XML document representations that jointly capture textual and structural information thereby incorporating both sources of evidence into a single similarity function.

Data Transformation

It is very rarely the case that data representing an entity is already available in a suitable format for matching. Thus, data transformation is crucial to EM: it can impact accuracy by alleviating data heterogeneity and efficiency by reducing the data or converting it to a simpler representation. For example, consider using a phonetic encoding algorithm, say, Soundex. While exposing spelling variations, Soundex converts data from the string data type to the integer data type, and, thus, reduces similarity calculation between strings to a simple equality test between integers.

The example above also illustrates that the distinction between data transformation and similarity functions is often blurred. In fact, sometimes the transformation applied to the data is the most important part of a similarity measure; the subsequent computation to obtain a similarity value is a secondary operation. This is the case of the class of token-based similarity functions, which employs a tokenization transformation as suboperation. Furthermore, other approaches to similarity are conceptually connected with transformation operation; for example, the similarity, or distance, of two objects can defined in terms of the operations needed to transform one object into another; a prime example is the family of edit-distance-based similarity functions. Section 2.3 is dedicated to these two important classes of similarity functions.

Data transformation can be a requisite for classification model design. For example, schema matching may be needed to uncover semantic correspondences among schema elements from different data sources before the selection of appropriate comparison fields. Transformations can take place in the evaluation phase, too. For example, they can be carried out jointly with matching, e.g., at query evaluation time in EII systems.

At this point, it should be clear that we use the term data transformation to generally refer to any transformation on the data that represents an entity including conversion of types (e.g., string to integer, scalar to sets, lists, or records), rescaling, normalization, discretization, interpolation, aggregation, syntactic and semantic substitution (e.g., substitution of characters and terms by corresponding synonyms), metadata modification, and so on; moreover, it also embodies other kinds of operations such as creation new data from old one, or selection of a subsets of data values (e.g., sampling). Next, we discuss these transformations in greater detail; we also include in our discussion *schema matching* and *schema mapping*, operations performed at the schema level that are a common precondition for further data transformation.

Text Segmentation This transformation converts flat strings into structured format using *Information Extraction* (IE) techniques [Sar08]. The type of the structure extracted is a set of *named entities*; examples of such entities are person names, streets, and companies. Text segmentation is important to EM due to several reasons: it prevents string comparisons among unrelated concepts, allows more granular field selection, and avoids distortion of statistics, i.e., a term can appear in several fields but with different frequencies. Address information is one the most common applications of text segmentation in EM because of the common practice of representing this sort of information as unstructured text and the absence of widely adopted standards.

As for other IE tasks, techniques for text segmentation evolved from rule-based approaches with manually coded rules to statistical learning models, where the rules are automatically learned from examples [Sar08]; the state of the art of this latter approach is based on Conditional Random Fields [LMP01].

Schema Matching Together with schema mapping, schema matching precedes EM in many data integration scenarios; in other words, *schema-level heterogeneity* is addressed before *instance-level heterogeneity* (however, see [BN05] for an opposite approach where EM is used to support schema matching). The foundational operator in schema matching is *Match* [RB01], which takes two schemas as input and returns a set of mappings between semantically corresponding elements. Schema matching techniques exploit syntactical similarity, element types, schema structure, data values, and prior matching information, frequently combined in a *hybrid* or *composite* implementation of Match [RB01]. Besides, schema matching applications are often furnished with visual tools to support user interaction. In this context, *model management supporting tools* are essential to harness the inherent complexity of the metadata produced by schema matching activities [Gör10].

Schema Mapping The correspondences returned by the Match operator are *uninterpreted* [HHH$^+$], in the sense that they do not readily allow *data exchange* or query answering. Therefore, schema mapping is necessary to produce interpretations of schema element correspondences, i.e., specifications that logically describe relationships between schema elements and corresponding data values. Such specifications capture information needed for the generation of physical artifacts that will ultimately perform data transformation. Schema mapping applications usually contain a schema matching component used to generate an initial set of mappings and a visual tool to refine existing mappings or to create new ones.

Simple Value Transformation This class of transformations comprises elementary data transformations, including fill-in of missing information and simple data consistency correction (e.g., by using knowledge of functional dependencies), and data domain transformation. This latter transformation typically aims at conforming simple data values from different sources, improving efficiency, e.g., mapping strings to their *fingerprint* hash values, or tailoring data values to a specific similarity function. For instance, after performing text segmentation, the string representation of the named entity *year* can be converted to a numerical data type; afterwards, the absolute difference can be employed as a similarity function.

Character-level Transformation This transformation syntactically standardizes some characteristics of the string data at the character level; the aim of this transformation is to increase (decrease) the similarity of true field (mismatches) matches by removing trivial and semantically unimportant data discrepancies. Common examples are case conversions, removal of punctuation marks, and substitution of whitespaces by a special symbol. It also includes character normalization according to language-specific rules; for example, removal or substitution of diacritical marks (e.g., removal of accents, substitution of umlaut-vowels by vowel plus *e*), and spelling standardization (e.g., substitution of double *s* by *scharfes S*.

Tokenization This operation converts an object to a set or multiset (or bag) of tokens. What constitutes a token is dependent of the object of interest and the corresponding similarity function. We discuss tokenization methods in Section 2.3.

Non-syntactical Transformation Most string similarity functions exploit syntactical commonalities (or differences) to ascertain their similarity. However, two strings can still represent the same real world entity even if there is no or very little syntactical clue. It happens, for example, in comparisons of synonyms (e.g., *Kaiserslautern* vs. *K-town*) or involving abbreviations (e.g., *Kaiserslautern* vs. *KL*). In such cases, similarity functions based on syntactical matches (or mismatches) would underestimate the true similarity between two strings. Conversely, polysemous words, i.e., words with multiple meanings, may lead to overestimation of the similarity.

The identification of the correct meaning of a word relates to the general problem of Word Sense Disambiguation (WSD) [Nav09]. At the schema level, WSD methods can be used to implement Match, i.e., as part of a schema matching process. For example, the work in [MMR05] disambiguates tag labels of XML documents by exploiting structural context (i.e., ancestor, descendants, and siblings) and external knowledge sources. At the instance level, the semantics of data values is normally constrained by the corresponding schema element. As a result, semantic variations such as polysemy are not expected, or restricted to abbreviations (e.g, *L.* can be an abbreviation to *Leonardo* or *Lucas*, *T.* can be an abbreviation to *Tom* or *Thomas*). Also, domain-specific knowledge sources, such as address formats or organization acronyms, are predominantly used, in contrast to general, lexical-oriented external sources, like the WordNet [Fel98]. Alternatively, one can learn the set of transformations from labeled data [ACK09].

Given a set of semantic correspondences, the next step is to use these correspondences to carry out the actual data transformation. At the schema level, this task is performed by schema mapping operations. Correspondences between instance-level values can be straightforwardly used in the pre-processing phase by converting all variations of a string to a specific canonical representation. However, as observed in [ACK08], this approach cannot adequately handle polysemy (as caused by abbreviations). In contrast, transformations can be performed in the evaluation phase: in [ACK08], the value of a string is expanded into the set of strings derived from it according to the given semantic correspondences; the similarity of two strings is then calculated in terms of their expanded sets.

Data Reduction When dealing with large datasets, a natural approach is to employ data reduction techniques. Examples of popular reduction techniques are histograms, sampling, Singular Value Decomposition (SVD), and Discrete Wavelet Transform (DWT) [BDF+97]. When the objects of interest are represented by collections of features (e.g., tokens), high dimensionality of the feature space is a critical problem. Besides making multi-dimensional search structures ineffective [WSB98], high-dimensional spaces exhibit poor discrimination among objects for popular measures, e.g., the Euclidean distance, which raises the question of whether or not similarity-based operations are meaningful in such situations [BGRS99]. Fortunately, real-world datasets very often present strong correlations between various dimensions or contain unimportant dimensions (e.g., those not exhibiting any discriminating power) thereby effectively reducing the *intrinsic dimensionality* compared to the dimensionality of the underlying space. In such cases, data reduction methods (e.g., SVD and DWT) and embedding methods (e.g., FastMap [FL95]) can be employed to uncover the intrinsic dimensionality of a dataset. Of course, these methods may cause accuracy degradation whenever the resulting dimensionality of the reduced space is lower than the intrinsic dimensionality. Some embedding methods are inherently approximate, but allow to bound the probability of incorrect results. The prominent example is Locality Sensitive Hashing (LSH) [GIM99], which is based on hashing functions that are approximately distance-preserving with probabilistic guarantees.

It is not always the case, however, that dimension reduction hurts accuracy. Some methods are able to discover hidden semantic structures in text data such as synonyms and polysemes. Latent Semantic Indexing (LSI) [DDL+90] uses SVD to construct a linear feature space that approximates the original space while capturing most of its variance; LSI is able to handle synonymy. Other approaches based on generative probabilistic models such as Probabilistic Semantic Indexing (pLSI) [Hof01] and Latent Dirichlet Allocation (LDA) [BNJ03] are able to handle synonymy as well as polysemy. Unfortunately, all these approaches that based on some form of spectral decomposition are computationally expensive and therefore do not scale for large datasets.

In contrast to constructing a new feature space, some techniques reduce dimensionality by selecting a subset of the original feature space. Such techniques can achieve aggressive dimension reduction, often reducing the feature space by a factor of about 10. In an extensive comparative study in the context of text categorization, Yang and Pedersen [YP97] found techniques based on document frequency (DF), information gain, and χ^2 test the most effective. Interestingly, the good result of the DF technique, which simply retains features — in this experiment, features are text terms — that occur in the highest number of documents seems to contradict the intuition that features with *low frequency* are most informative; as already mentioned, this intuition is behind the approach of Newcombe et al. and the widely used IDF weighting scheme. However, as observed in [Seb02], it is well known that the term frequency distributions are characterized by very large numbers of extremely rare terms [Baa01]; according to the reduction factor reported in the experiments of [YP97], most of the features removed are indeed such terms. In the EM context, for some field values, very rare features indicate data entry errors such as typos and misspellings, and can be ignored.

Stopping and Stemming These operations are widespread in IR [WMB99]. *Stopping* is a data reduction technique[7] which removes certain frequently occurring terms. As such, stopping could be seen as the opposite of the DF technique for feature selection. However, high frequency alone is not a reliable condition to deem a term as unimportant [DFS+09]. For example, some terms such as *computer* and *company* can be highly frequent in a document collection yet convey useful semantic information. Moreover, some stop terms in one domain may not be a stop term in another domain. Therefore, in practice, lists of stop terms have to be manually defined by an application domain expert. Recent work has addressed the problem of automatically finding the list of stop terms in the context of Web query interface integration, which was shown to be NP-complete [DFS+09].

Stemming removes one or more suffixes of a term to obtain the corresponding *stem*, i.e., the root form of the term without morphological constructs derived from grammatical categories, such as tense and plurality. For example, consider the terms *adapt*, *adapted*, *adapting*, *adaptation*, *adaptations*. These terms can be conflated into the single term *adapt* by a stemming procedure. The most popular stemming algorithm is the Porter's algorithm [Por80].

[7]We describe stopping here due to its historical connection to stemming.

XML-structural Transformation For XML datasets following multiple schematic rules, schema matching operations can be used to align the underlying structure before EM takes place. However, this approach is hardly feasible for large, non-schematic XML datasets. While it is possible to tackle this problem at query processing time — as we propose in this thesis — pre-processing transformations, such as hierarchy restructuring, can nevertheless be employed to ameliorate the structural heterogeneity. Some algorithms for XML-structure pre-processing are presented in [GdSM07].

Blocking

Scalability is a serious concern in EM tasks. A naive approach compares every pair of entities, which leads to a prohibitive time complexity of $O(n^2)$ on databases containing n entities. Thus, reduction of the comparison space is mandatory to tackle large databases [NKAJ59]. The general approach is based on a two-step strategy: candidate pairs are generated by bringing together potential duplicates using an inexpensive operation while discarding the maximum number of (hopefully) irrelevant candidate pairs; afterwards, more accurate (and expensive) operations are carried out to identify duplicates. A similar strategy is adopted by set similarity join algorithms as we will study in detail in Chapter 5.

Candidate generation in EM systems is performed by blocking methods. The general idea is to divide the data into (possibly overlapping) blocks and consider only records within the same block for candidate generation. Typically, blocking methods apply sorting or simple similarity functions on a subset of the comparison fields. Clearly, it may hurt accuracy when true duplicates do not fall into the same block. Multiple blocking operations can be performed to avoid or at least alleviate accuracy degradation, albeit at the expense of increased runtime. This can be done by using multiple comparison fields, similarity functions, or both. In general, there is an intrinsic trade-off between performance and accuracy in the design of blocking methods and, in favor of accuracy, many criteria considered in Classification Model Design can be applied to Blocking as well.

A popular blocking method is the *sorted neighborhood* [HS98]. This method first extracts keys by applying a function provided by a domain expert on one or more record fields. An example of such a function is the concatenation of the first three letters of the field `Name` and the first three consonants of `City`; all letters are converted to uppercase and punctuation marks removed. Therefore, the set of keys for the sample data in Figure 2.1 is {`LEOKL`,`LEOKSR`,`LARKSR`}. The records are then sorted according to the key generated and a window of fixed size w slides over the the data; every new record entering the window is compared with the previous $w - 1$ records. Thus, the number of comparisons is reduced from $O(n^2)$ to $O(wn)$ for a database containing n records. The authors empirically demonstrated that small windows provide superior accuracy results.

Another widely used blocking method consists in using a cheap similarity function to divide the data into overlapping subsets called *canopies* [MNU00]. To create the set of canopies, one starts with a list containing all records and two similarity thresholds, τ_1 and τ_2, where $\tau_1 < \tau_2$. Then, a record r is randomly selected from the list and the similarity between r and the other records of the list is calculated. Further, a new canopy is created and r and all records within similarity τ_1 are put in it; all records within similarity τ_2 and r are removed from the list. These steps are repeated until the list is empty. The estimated number of comparisons using canopies is $O(f^2n^2/c)$, where n is the number of records, c is the number of canopies, and f is the average number of records per canopy. A comparison between canopies and sorted neighborhood as well as other blocking methods is presented in [BCC03].

Recent work on blocking methods includes: adaptation of sorted neighborhood to XML data [PWN06], learnable blocking functions [BKM06], and interactive blocking [WMK+09]. In the latter, blocking is performed jointly with merging of duplicates and block instances communicate their matching results in an interactive fashion.

Matching and Classification

Matching and Classification constitute the essence of an EM task. Matching uses a set of similarity functions to compare the candidate pairs delivered by the blocking method; the comparison results are then fed to Classification, which classify the corresponding record pair as duplicate, non-duplicate, or as requiring further analysis. We have already covered the most important factors affecting the accuracy of these components when we discussed the Classification Model Design. Now, we focus on implementation issues and efficiency aspects. Again, many techniques described here are also applicable to Blocking.

The implementation of Matching and Classification is largely dictated by the formulation of the decision rule and, in turn, by the strategy for combination of matching results. In this regard, the evaluation can be performed stepwise or "one shot", interleaved or strictly sequential; some approaches provide more scope for optimization, as already mentioned.

Several classification models used in EM tasks employ decision rules that can be interpreted or easily converted to a propositional expression composed by a set of disjuncts and conjuncts on similarity functions [CNS04, CCGK07]. D-trees and rule-based classifiers are examples of such models. Furthermore, classifiers like naive Bayes, which combine similarity results in more intricate ways, can still be post-processed to extract simpler predicate expressions [CNS04]. Such formulations are appealing because they lend themselves to leveraging traditional query optimization techniques for the derivation of pipelined execution plans and exploitation of indexes (e.g., functional indexes) [SK03, CNS04, CCGK07]. Of course, processing of conjunctions can always be promptly interrupted whenever the evaluation of a simple similarity predicate returns false, thereby saving similarity calculations. The flip side is that the derived propositional expression can be very complex. The problem of identifying better execution plans for arbitrarily complex propositional expressions is not trivial [CGS03].

Another popular decision rule formulation is given by a thresholded linear combination of the form $\sum_i w_i sf_i \geq \tau$, where w_i is the weight, possibly negative, associated to the similarity function sf_i. This formulation is adopted, for example, by SVM models [BM03a] and approaches employing combination of evidence based on regression analysis [Cro00]. In contrast to propositional expressions, such formulations require applying the decision rule on every candidate pair [CCGK07]. Notwithstanding, when comparing two records, one can keep track of the similarity upper bound after each field comparison and terminate the evaluation as soon as this upper bound is less then the similarity threshold. Other combination approaches demand the calculation of all similarity values. This is the case for methods using standard aggregate functions such as *min* and *max*. Finally, combination techniques that exploit ranking positions or scored ranking values require, besides the calculation of all similarity values, the matching result between a given record and all other records associated with this record in the candidate set before applying the decision rule.

Some similarity functions are computationally more expensive than others. On the other hand, owing to the threshold value and the similarity function itself, some similarity predicates are much more selective than others. Therefore, evaluation order of similarity predicates may have a significant performance impact, especially for decision rule formulations allowing earlier termination. In this scenario, we can borrow techniques from the work on expensive-predicate query optimization. A well-known optimization consists of determining the evaluation order of predicates according to the following metric [HS93]:

$$rank_p = \frac{selectivity(p) - 1}{cost(p)} \tag{2.5}$$

where $cost(p)$ and $selectivity(p)$ are the cost of evaluation and the selectivity of a predicate p, respectively. In our case, the predicate p involves a similarity function. The relative cost of a similarity function can be defined in several conceivable ways, for example, based on the asymptotic cost or average execution time. Also, as EM tasks typically involve bulk processing, dataset characteristics are likely to be a determining factor for the overall processing cost. Furthermore, a great deal of work has addressed the selectivity estimation of similarity operations, see, e.g., [HYKS08], [LNS09], and references therein.

EM tasks are inherently domain-dependent; moreover, EM is often not executed in isolation, but as part of a more complex data cleaning or data integration solution. Hence, a crucial aspect for domain- and application-neutral EM systems is the accommodation of a variety of matching operators and classification logic in a general and flexible framework. In this direction, Galhardas et al. [GFS+01] presented *Ajax*, a declarative data cleaning framework, which allows to decouple the logic specification of EM operators from their physical implementation; Ajax has been extended to the *Xclean* project to support XML data [WM07]. Following a more fine-granular approach, Chaudhuri et al. [CGK06] designed a primitive similarity operator that can be composed with other traditional query operators (e.g., relational operators) to support similarity joins based on several similarity functions. In the ALIAS system [SB02, SK03], the evaluation engine is based on *groupwise* operators, i.e., similarity operations can be performed on linear groups of records instead of just one pair of records at a time.

Most of the approaches discussed above are based on query processing techniques proposed in a DBMS context: pipelined query execution plans, predicate reordering, exploitation of indexes, cost models, selectivity estimation, and decoupling between logical and physical operators. Moreover, a very large body of work is available on similarity algorithms for operations such as joins and selections [Coh98, GIJ$^+$01, SK04, ABG05, CGK06, AGK06, BMS07, ABDG08, HCKS08, LLL08, XWLY08, XWL08, XWLS09] and, recently, Silva et al. [SAA10] studied transformation rules for the optimization of logical query plans containing similarity operators. All these contributions are geared towards the evaluation of EM tasks, in particular similarity operations, within the query engine of a DBMS, which is also the direction of our work in this thesis. We shall revisit this topic in Chapter 6, when we address the integration of our similarity join framework into an XDBMS.

Clustering

The relation "is duplicate of" is an equivalence class, i.e., it satisfies the properties of reflexivity, symmetry, and transitivity. Thus, clustering of duplicates forms a *partition* of a database — note that EM can be interpreted as a clustering task that groups duplicate entities. Following this premise, previous work has used single-link clustering [JD88], which is equivalent to calculating the transitive closure, to obtain the groups of duplicates [ME97, HS98, BM03a, WN05]. The most common implementation approach utilizes a union-find data structure to efficiently maintain the collection of disjoint updatable sets of duplicates [ME97]. The underlying assumption of this approach is that the corresponding classification model is able to perfectly identify all pairs of duplicates or, at least, it will produce no *false duplicates*, i.e., pairs erroneously classified as duplicates. Of course, such an assumption is overly unrealistic. In practice, clearly inconsistent results are commonly produced like classifying the pairs $\langle r_1, r_2\rangle$ and $\langle r_1, r_3\rangle$ as duplicates but not the pair $\langle r_2, r_3\rangle$, which violates the transitivity property. In the presence of such inconsistencies, the calculation of the transitivity closure leads to a propagation of false duplicates by placing unrelated records into a same group. Indeed, single linkage clustering has been shown to perform poorly in several experiments [BBS05, HCML09].

Avoiding the drawbacks of the transitivity assumption, Chaudhuri et al. [CGM05] identified two criteria for defining groups of duplicates, namely, the *compact set* and *sparse neighborhood* criteria. Similarly to the complete-link clustering methods, compact sets are defined in terms of *cliques* in a similarity graph, i.e., duplicate records in a group must be closer to each other than to other records. The sparse neighborhood criterion requires that a cluster of duplicates has no or only very few records in its immediate vicinity. The authors claim that these criteria capture structural properties characterizing clusters of duplicates in real scenarios.

The problem of minimizing inconsistencies in the output of an EM process can be formulated as the Correlation Clustering problem [BBC04]: given a complete graph with edges labeled "+" or "−", find a clustering of the vertices that minimizes the number of "−" edges inside clusters and the number of "+" edges between clusters. An EM output can be easily modeled according to this problem by connecting record pairs in D with "+" edges and pairs in N with "−" edges; additionally, pairs in H can be given a "0" label or left unconnected. Besides its theoretical soundness, Correlation Clustering is attractive because it does not require the specification of extra parameters, such as the number of clusters for partitional clustering or the cutting threshold of the dendogram for hierarchical clustering. Unfortunately, finding an optimal partitioning for general graphs is NP-hard. Moreover, although there exist approximation algorithms for this problem, such solutions are still prohibitively expensive. A practical alternative is to follow a divide-and-merge strategy as proposed in [CKVW06]. In the divide phase, a divisive hierarchical algorithm is used to produce a tree whose leaves are records. In the merge phase, records are grouped using the correlation clustering as objective function.

Finally, an extensive evaluation of clustering algorithms in the EM context is presented in [HCML09]. Unfortunately, the only *Agglomerative Hierarchical Clustering* (AHC) method evaluated was the single-link method. Other popular AHC methods such as complete-link and UPGMA were not included in the evaluation. Complete-link and UPGMA were found to produce accurate results in [BBS05], which meets our own experience (see Chapter 3).

2.1.3 Recent EM Approaches

There has been a flurry of recent work on EM motivated by the increasing interest in related application areas, such as information extraction, management of uncertain and inconsistent data, and dataspace systems, as well as by recent progress in key technologies such as statistical relational learning. Thus far, our discussion has been based on the traditional NFS framework (see Figure 2.2). Not all approaches considered in this Section freely adhere to the NFS framework, however. For this reason, whenever pertinent, we point out adjustments for accommodation of the corresponding approaches into the NFS framework in the following discussion.

Merge-and-Refine Strategy

A natural step after EM is the merging of the identified duplicates into a single representation. [8] In contrast to performing identification and merging of duplicates sequentially, recent work has followed a merge-and-refine strategy, where matching and merging of duplicates are performed in a combined and iterative fashion [BGMM$^+$09]. The key insight is that record merges can lead to further matches.

For example, consider the sample data in Figure 2.1. The output of a hypothetical EM task could be the pair formed by the first two records, i.e., $\langle r_1, r_2\rangle$, because of the

[8] This step is commonly referred to as *data fusion* [BN08]. Note that the term data fusion is also used to refer to rank-based combination of evidence in the IR literature. Therefore, to avoid confusion, we use here the term duplicate merging or simply merging.

high similarity of their field Name. The third record, r_3, does not match either r_1 or r_2 because its field Name is not similar to that of the other two (due to the different naming convention). Now, consider a *merge function* that adopts the policy of picking the most informative (and misspelling-free) field representation between two records to construct a new merged record. Hence, the result of the merging of r_1 and r_2 would be the new record r_{12}:

r_{12}	Leonardo Ribeiro	Kaiserslautern	111-2322	Gottlieb-Daimler-Straße

Note that r_3 may now match with r_{12} because of the increased similarity on the fields City and Address.

Characteristics of the functions used to identify and merge duplicates may affect the performance significantly. Benjelloun et al. [BGMM+09] identified the so-called *ICAR* properties — namely, indempotence, commutativity, associativity, and representativity — and developed algorithms to exploit them. In their work, the authors viewed the functions used to merge and compare records as ″black-boxes“ without considering their internal details; it was only assumed that some ICAR property holds or not, for a given function. Drawing a parallel with the NFS framework, Matching and Classification are performed in a single component producing a simple yes-no output, which is used to decide whether to merge two records or not. Note that the record merging function can be rather complex. For instance, it is not always clear how to decide which value to choose in case of conflicting data or to identify the most informative representation (e.g., see the Phno attribute in Figure 2.1).

Collective Matching

All approaches discussed so far compare and classify two records at a time. Moreover, pairwise decisions are made independently, i.e., a pair of record is classified as duplicate or non-duplicate regardless of all other records in the database. Even when using the merge-and-refine strategy describe above, positive duplicate decisions only affect other decisions implicitly: first, a record is constructed from a duplicate pair by the merge function; then this record is compared anew with all the other records.

Frequently, the information about entities spans multiples fields that, in turn, also represent matchable entities on their own. In such cases, a different approach is to model the interrelationships of classification decisions *explicitly* and automatically propagate results between related entities. As a result, the classification is performed *collectively* on a group of entities. For example, consider again the sample database in Figure 2.1; each entity represented by a record can be seen as composed by the sub-entities Name, City, and Address. Classifying r_1 and r_2 as duplicates forces the cities ″KL“ and ″Kaiserslautern“ and the addresses ″Gottlieb-Daimler-Straße“ and ″Gottlieb Str.“ to be called duplicates, too, even though the corresponding textual similarity is not high. In turn, the belief that r_3 represent the same entity as r_1 and r_2 is now increased accordingly.

Most relevant work on collective matching is based on sophisticated probabilistic models that represent interrelationships of classification decisions as random variables and employ an inference engine to conduct the EM task. Examples of such probabilistic models include relational Bayesian [PMM$^+$02], Markov logic [SD06], Conditional Random Fields [MW04], and Latent Dirichlet Allocation models [BNJ03].

In comparison to the NFS framework, collective matching combines Clustering with Classification. Note that the result may still be inconsistent, e.g., not satisfying the transitivity property. Thus, an additional clustering step as discussed previously is however necessary.

Incorporating User Feedback

Often, the set H generated at the end of the classification process is very large. This happens, for example, when the cost of misclassification is high. Consider an HN system correlating individual medical information. Because of the disastrous consequence of mixing information of different individuals, EM tasks in such scenarios prudently adopt a very conservative classification model that precludes automatic matching even for candidate pairs exhibiting high evidence for a decision. As a result, it can be overwhelming for the user to inspect all or even a large part of the candidate pairs. Hence, it is essential to select elements of H for presenting to the user in a reasonable order.

In a dataspace system context, Jeffery et al. [JFH08] proposed a decision-theoretic framework based on the *value of perfect information* (VPI) for ordering candidate pairs for user confirmation. Informally, VPI assess the benefit of information by measuring how much the state of a system improves after acquiring this information; the state of the system is measured by a utility function. In the case here, the information corresponds to the true matching status of a candidate pair provided by the user. The utility function is defined in terms of the effectiveness of the dataspace in answering a pre-defined query workload. There are two components in the utility calculation: recall and query importance. Recall refers to the completeness of the result of a query with respect to a perfect dataspace, i.e., a dataspace without duplicates; the importance of a query is given by a weight parameter, e.g., derived from the frequency of the query in the system. The authors made several simplifying assumptions to approximate the associated VPI value of a candidate pair. The VPI formulation as well as its approximation is tightly tied to the weighting scheme applied to the query workload. To use this technique in general EM systems, it is possible to attach weights to the entities of H. For example, such weights can be based on the number of candidate pairs in which an entity appears. An interesting question is whether it is possible to obtain utility gains without explicitly assigning weights to entities.

User feedback strategies are complementary to the NFS framework: they can be straightforwardly invoked between the Classification and Clustering at each interaction step. In fact, the approach of Jeffery et al. can be seen as the counterpart for the evaluation phase of interactive approaches for decision model design like [SB02] and [TKM02]. In a related work, Chai et al. [CVDN09] presented a user feedback framework, including a declarative language, that allows developers to write complex user interaction logic at various stages of an EM workflow.

Constraint Repair Model

The presence of fuzzy duplicates violates integrity constraints. For example, consider the functional dependency $[\texttt{Phno}] \rightarrow [\texttt{City}, \texttt{Address}]$ in the database shown in Figure 2.1. This functional dependency is violated by the duplicates r_1 and r_3. Repairing such a constraint involves identifying r_1 and r_3 as duplicates, which is equivalent to an EM task. The repair of an inconsistent database is another database that satisfies integrity constraints and differs minimally from the original database [ABC99]. Thus, duplicate identification can be modeled by specifying a set of constraints on a database and then finding a *repair* [BFFR05].

Operations used to derive a repair are record insertion, record deletion, and value modification. Bohannon et al. [BFFR05] proposed a repair cost model based on data accuracy — represented by user-defined weights — and similarity between the original value and the repair alternative; value modification operations are relatively inexpensive whenever the accuracy weight of the original value is low and the similarity is high. The motivation behind this approach is that value modification is preferable to record insertion or deletion in common situations, for example, to avoid loss of information due to record deletion. Unfortunately, finding a min-cost repair when considering value modifications is NP-complete in the size of the database [BFFR05]. Bohannon et al. proposed a heuristic based on equivalence classes of attribute values; further performance improvements are obtained by using a number of optimizations such as blocking methods [HS98].

As mentioned before, constraints can also be exploited to improve accuracy. Chaudhuri et al. [CSGK07] used groupwise aggregation constraints such as $\texttt{sum}(\texttt{Billed}) = \texttt{sum}(\texttt{Shipped})$ and modeled the EM task as a constraint satisfaction problem [RN03]. To avoid intractability while incorporating textual similarity into the optimization framework, the authors constructed a dendogram by applying a hierarchical clustering algorithm (e.g., single-linkage) over the set of entities and restricted the groups of candidate duplicates to those obtained by different cuttings of the dendogram. In this way, the approach of Chaudhuri et al. can be viewed as performing Clustering prior to Classification.

Traditional constraint repair approaches rely on schema-based dependencies such as inclusion and functional dependencies to detect data inconsistencies. Such dependencies are often "too loose" to capture subtle data quality issues such as fuzzy duplicates. Bohannon et al. [BFG+07] introduced the concept of *conditional function dependencies* (CFDs) extending the expressiveness of standard functional dependencies to capture semantic relationships between data values as specified by a *pattern tableau*. For example, CFDs can express dependencies of the form $[CountryCode = 49, ZIP] \rightarrow [City, Adress]$, which states that for Germany ("Country Code = 49"), the zip code determines city and the address. It has been empirically shown that CFDs are indeed more effective than standards dependencies in repairing dirty data [CFG+07].

Probabilistic Databases

The constraint repair model described above produces a "clean" version of the original database where all identified duplicates have been merged into a single representation. This clean version is promptly available for use through any queryable system such as regular DBMSs and search engines. This approach is referred to as *one-shot* cleaning approach. Despite the readiness, one-shot cleaning can be unsatisfactory due to the following reasons:

- Loss and degradation of information: EM as well as duplicate merging are intrinsically imprecise operations even when conducted by semi-automatic processes. After carrying out one-shot cleaning, unless there is support for *data lineage* [BSH+08], there is no information linking the conciliate representation of an entity and the corresponding duplicates. This means that any error in the cleaning operation like lossy merging or erroneous duplicate classification is unrecoverable. As a consequence, data accuracy continuously degrades as cleaning operations are performed.
- Non-updatable decisions: This issue is closely related to the first problem. Consider that records r_1 and r_2 are classified as duplicates by an EM process with confidence c_1 (e.g., given by the similarity score). Later, a new record, r_3, is inserted into the database. The same EM process classifies r_2 and r_3 as duplicates with confidence c_2, where $c_2 > c_1$; however, r_3 is assessed far away from r_1. Intuitively, this latter result decreases the confidence in the first matching decision about r_1 and r_2. But, if r_1 and r_2 are merged in the first classification into a new record, it could be impossible to identify and exploit the new information originated from the insertion of r_3 and refine the previous matching decision. The situation is similar for any kind of new information such as filling of a missing field or user feedback. In general, one-shot cleaning makes matching decisions produced by an EM system permanent.
- Strong dependency on a single EM strategy: Different EM strategies, e.g., procedures, algorithms, parameter settings, are likely to produce different results. Identifying the best EM strategy for an application scenario demands an exorbitant amount of tuning effort, if possible at all. Moreover, because the very notion of duplication is context-sensitive, the best EM strategy is expected to vary accordingly acrross application scenarios. Therefore, instead of producing a single database instance as for one-shot cleaning, it may be desirable to use multiple EM strategies to obtain several clean versions of the original database.

An alternative to avoid the shortcomings of one-shot cleaning is to keep all duplicates and use a *probabilistic DBMS* [DRS09] to manage the dirty database. Duplicates can be interpreted as alternative representations of a real-world entity [AFM06]. In this sense, a dirty database represents multiple possibly clean databases; each clean database is derived from different EM and merging results. If we associate probabilities to each record of the dirty database, the connection of this interpretation to the *possible world semantics* pervasively used by probabilistic DBMS is straightforward. Furthermore, because a database containing duplicates is inconsistent, querying such database is also strongly related to the problem of obtaining consistent query answers from inconsistent databases [ABC99]. In fact, a consistent answer corresponds precisely to an answer in a probabilistic database that has a probability of 1 [AFM06].

Andritsos et al. [AFM06] modeled dirty databases as disjoint clusters of duplicates. Clean databases are defined by selecting one record from each cluster. A value is associated with each record that quantifies the probability of this record being selected to be in the clean database (for non-duplicates the probability is 1, of course). Note that, in this way, records from the same cluster are mutually exclusive and records from different clusters are independent. This setting is known as the *block independent-disjoint* representation model [DRS09]. The probability of a candidate clean database is defined as the product of the probabilities of each of its records. For each cluster, record probabilities are calculated by first building a *cluster representative* and then measuring the similarity between each record and the cluster representative. The intuition is that records that are closer to the cluster representative are more likely to be in the clean database. Query answering over the dirty database is carried out by rewriting the original query into another query that returns answers along with the probability of the answers being in the clean database.

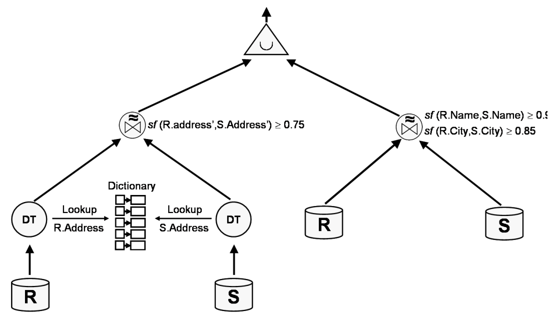
Beskales et al. [BSIBD09] extended the approach of [AFM06] by modeling the results obtained by different parametrization of the clustering algorithm. Specifically, the authors employed a hierarchical clustering algorithm and obtained distinct set of clusters by cutting the dendogram at different thresholds.

Cheng et al. [CCX08] introduced the *PWS-quality*, which employs a entropy-based measure for quantifying the uncertainty of the answers returned by probabilistic databases. In spirit similar to the work in [JFH08], the PWS-quality metric can be used to guide user feedback by identifying a set of records that, when cleaned, will lead to the highest quality improvement.

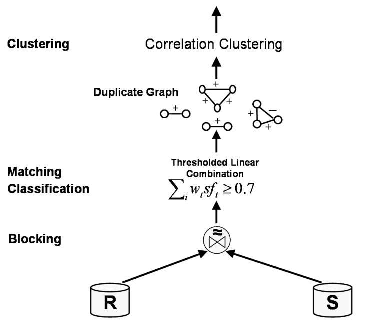
2.1.4 Similarity Join Use Cases

The above discussion on EM systems revealed a very complex landscape containing many alternative, sometimes contrasting, approaches. This diversity is partially justified by heterogeneous viewpoints of the research communities which have been involved with the EM problem. Another explanation relates to the fluctuating nature of the concept of duplication: a pair of entities can be duplicates in one context, but not in others [ZB06]. For example, two records can be interpreted as spurious duplicates in a data warehouse, but as legitimate representations of multiple versions of an entity in a revision control system. Hence, it is very unlikely that a single EM strategy will be universally effective across all scenarios and the diversity of approaches is rather a demand for handling the manifold facets of the duplication problem.

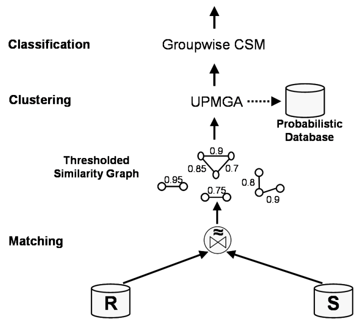
Despite the considerations above, it is still possible to distinguish commonsense principles. In particular, the notion of similarity is ubiquitously employed as a fun-



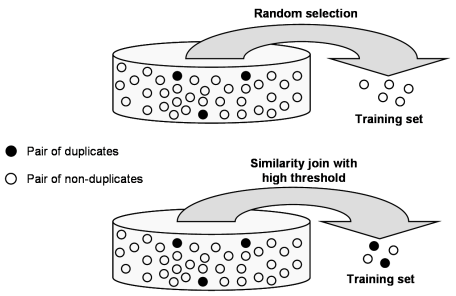
(a) DNF query plan



(b) Similarity join as Blocking component



(c) Similarity join as Matching component



(d) Training set construction

Figure 2.3: Similarity join applications in a variety of EM scenarios

damental device for identifying duplicates. As a result, joining entities that are similar is an operation of prime importance in EM systems. Further, because EM only becomes a relevant problem in face of massive datasets, supporting large scale similarity processing is fundamental. Thus, it is not a surprise that similarity joins are the main workhorse of all modern EM systems.

We now emphasize the importance of similarity joins with examples drawn from the EM strategies previously discussed. Figure 2.3 illustrates four different example EM scenarios, which we discuss in the following.

Example 2.1. ***Query plans***: *A query plan composed by similarity joins and other operators is shown in Figure 2.3(a). This example is based on the operator tree from Chaudhuri et al. [CCGK07], which implements EM decision rules formulated in disjunctive normal form (DNF). The plan basically consists of the union of two similarity joins. (Note that we use the symbol* $\overset{\approx}{\bowtie}$ *to represent the similarity join operator.) Additionally, the left subtree contains data transformation operators using external knowledge sources (a table of address synonyms). Similar operator trees can be used to implement other kinds of decision rules represented as propositional expressions accordingly [SB02, SK03].*

Example 2.2. ***Blocking operator***: *Figure 2.3(b) illustrates the use of similarity joins for blocking. The decision rule is now formulated as a thresholded linear combination as adopted* *by SVM and logistic regression models [BM03a, PC98]. Such formulations do not lend themselves to effective reduction of the comparison space. Similarity joins using an inexpensive similarity function can be used to avoid performing the linear combination model over the cross product of the two inputs. The output of Classification is a duplicate graph, on which the Correlation Clustering algorithm is applied to correct matching inconsistencies.*

Example 2.3. ***Matching operator****: In Figure 2.3(c), the similarity join operator implements the Matching component where the similarity join is used to construct a similarity graph. Following the evaluation course, a hierarchical clustering algorithm is used to construct clusters of duplicates that serve as input to a groupwise constraint satisfaction model as proposed in [CSGK07]. Alternatively, the set of clusters can be used to create a probabilistic database [BSIBD09].*

Example 2.4. ***Training set construction****: In the previous three examples, similarity joins were used in the evaluation phase. Figure 2.3(d) illustrates their use to support classification model design. Now, the similarity join is employed to construct a training set for a learning approach. A meaningful training set for EM should contain representative examples of duplicates and non-duplicates. As observed in [SB02], it is often hard to find a challenging set of non-duplicates, i.e., unrelated records that are likely to be erroneously classified as duplicates. Typically, real-world datasets are highly skewed towards non-duplicates. As a result, a training set constructed by randomly selecting a pair of records will be constituted mostly by non-duplicates. A way to tackle this issue is employing a similarity join operator to complement the training set. By using a high threshold value, record pairs in the result are likely to be duplicates. At the same time, any identified non-duplicate is likely to be a more challenging example than a randomly selected record pair. This approach was proposed by Bilenko and Mooney [BM03b] as an alternative to active learning methods [SB02, TKM02].*

Effectiveness and efficiency are the principal concerns of similarity join algorithms. Efficiency aspects are addressed in Chapter 5. Effectiveness is dictated by the underling similarity function. We start our discussion on similarity functions in Section 2.3 and we describe how we will measure the result quality of similarity joins in Section 2.4.

2.2 XML Similarity Joins

Henceforth, we focus on XML data. Similarity assessment is especially difficult on XML datasets, because structure, besides textual information, may embody variations in XML documents representing the same real-world entity. This aspect brings a new quality dimension to the EM problem and, therefore, textual techniques developed for relational data are not sufficient. The following example illustrates this observation.

Example 2.5. *Consider the illustration in Figure 2.4, which shows an XML document fragment containing hypothetical patient medical information. Although subtrees a) and b) refer*

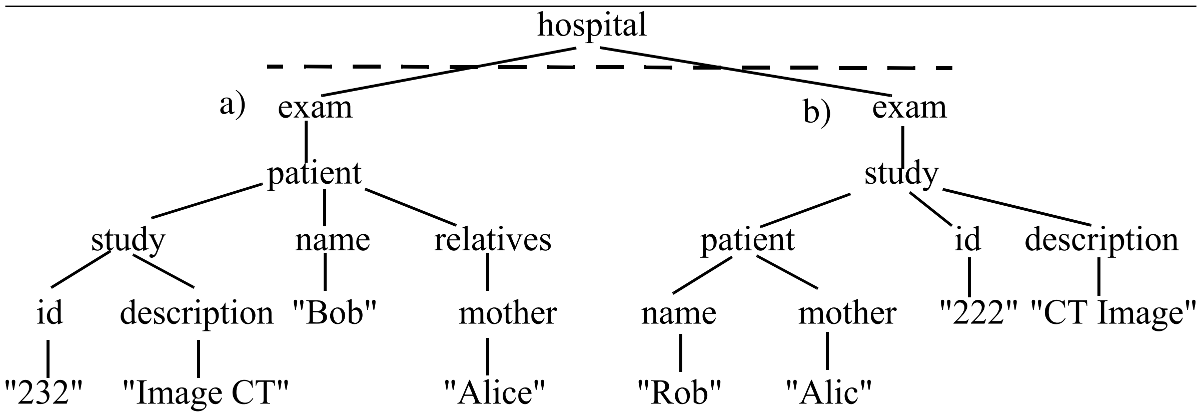


Figure 2.4: Example of a heterogenous XML document fragment

*to the same patient's exame, it would be extremely difficult for an EM application to correctly identify them as fuzzy duplicates. The data in subtree **a)** is arranged according to* `patient`*, while the data of subtree **b)** is arranged according to* `study`*. Further, there is the extra element* `relatives` *in subtree* a*). Moreover, there are typos (e.g., "Bob" and "Rob") and use of different abbreviation conventions between the content of the two subtrees ("Image CT" and "CT Image").*

Next in this section, we define our XML data model and introduce the related notation. Then, all assumptions about the input XML dataset are made explicit, before we formally define the XML similarity join problem. Finally, we discuss differing XML models adopted by related work.

2.2.1 XML Data Model

Following the common practice [BKS02, GJK$^+$06, ABDG08], we model an XML document as a rooted, labeled tree $T(V, E)$. Each node $u \in V$ is represented by a triple (i, l, c), where i is the *node identifier*; $l \in \Sigma$ is the node label, where Σ is a finite alphabet of string literals; $c \in S$ is the node's textual content, where S is an infinite set of string values. Let $\epsilon \notin \Sigma, S$ represent the *null symbol*. Every node has an identifier that is unique in the whole XML collection. Nodes can not have both label and textual content. Nodes with a label are *element nodes* ($c = \epsilon$); nodes with content are *text nodes* ($l = \epsilon$). A node with $l = \epsilon$ and $c = \epsilon$ is a *null node*. Given a node u, $\varsigma(u)$ corresponds to the label of u, if u is an element node, or to u's content, if u is a text node, or to ϵ if u is a null node. When no confusion arises, we use u to represent an (text) element node as well its (content) label; we also omit node identifiers. Given two nodes u and v, we say that $u = v$ iff both u and v are element (text) nodes and their labels (texts) are the same.

Edges in E induce a binary relation on V, where each pair $(u, v) \in E$ represents the parent-child relationship between two nodes $u, v \in V$; we say that node u is the *parent* of node v and v is a *child* of node u. A node with no children is a *leaf*; if a leaf node is an element node, we say that this element is empty. Text nodes are always leaves. The number of children of an element node u is its *fanout* u_f. A node can have only one parent and nodes with the same parent are *siblings*. There exists only a single *root node* in a tree T, denoted as $root(T) \in V$, which has no parent; we denote as $T(u)$ the (sub)tree rooted at node u. Let $S(T) \subseteq V$ be the subset of element nodes of V, and $C(T) \subseteq V$ the subset of text nodes. We say that $S(T)$ represents the *structural part* of T, while $C(T)$ represents the *content part*. The size of a tree T is the number of nodes it contains, i.e., $|T| = |V|$.

A sequence of nodes $\langle u_0, u_1, ..., u_n \rangle$, for $n > 0$, is a path of *length* n, if u_k is the parent of u_{k+1} for $0 \leq k < n$. Alternatively, we will use the XPath notation to represent paths. For example, $u_0/u_1/u_2$ is a path of length 2. The *level* of a node u is the length of the path from the root node to u. The *height* of a tree is the length of the longest path from the root node to any of its leaves. If there is a path from u to v in the tree of size $k > 0$, then we say that u is the *ancestor* of v at distance k (if $k = 1$ then u is the parent of v; if $k = height$ then u is the root node) and v is a *descendant* of u. Given two nodes u and v, $rel(u, v)$ returns their relationship, i.e., parent-child, ancestor-descendant, or sibling[9]. Furthermore, we use a functional notation to denote attributes of a node. For example, $level(u)$ denotes the level of u, $anc(u)$ its the set of ancestors, and so on.

We consider *unordered* and *ordered* trees. In unordered trees, only parent-child and ancestor-descendant relationships are relevant, i.e., the children of a node u form a set. In ordered trees, the left-to-right order among siblings is relevant, i.e., the children of u form a sequence. For ordered trees, two siblings are contiguous if their position from left to right is k and $k + 1$, respectively. Typically, unordered trees are used to model *data-centric* XML, whereas ordered trees model document-centric XML [ABDG08]. A preorder traversal of a tree visits parents before children, whereas a postorder traversal visits children before parents; for both traversals, siblings are visited in left-to-right order and each node is visited only once. For ordered trees, a total order on the nodes can be obtained by any tree traversal; in our model, the total order induced by a preorder traversal corresponds to the *document order* defined on the nodes of XML documents[10].

Note that we use a simplified XML document model: we distinguish between element nodes and text nodes, but not between element nodes and attribute nodes: each attribute is child of its owning element, sorted by name, and appearing before all element ”siblings. Finally, we consider only data of string type and disregard other node types such as ID/IDREF attributes and Comment.

Example 2.6. *Consider subtrees a) and b) in Figure 2.4. Both subtrees have root nodes labeled as* `exam`*. The structural part of subtrees a) and b) are* $S(a)$ ={`exam`, `patient`, `study`, `id`, `description`, `name`, `relatives`, `mother`} *and* $S(b)$ ={`exam`, `study`, `patient`,

[9]In Chapter 6, we will see that the relationship between two nodes can be freely obtained by using a node identifier scheme based on Dewey classification.

[10]In document order, element nodes are ordered according to the occurrence of their start tag in the XML document [BBC+07].

`name`, `mother`, `id`, `description`}—both sets are ordered according to preorder traversal; the content part is $C(a)$ ={"232", "Image CT", "Bob", "Alice"} and $C(b)$ ={"Rob", "Alic", "282", "CT Image"}, respectively. The height of both subtrees is 4. The leftmost path of subtree a) is `exam`/`patient`/`study`/`id`/"232" and of subtree b) is `exam`/`study`/`patient`/`name`/"Rob".

2.2.2 Data Assumptions

Our framework aims at supporting similarity joins on non-schematic, heterogeneous XML datasets. However, due to the semi-structured, self-describing nature of XML, data and metadata can be arranged in very complex and unpredictable ways, even in our simplified data model. Moreover, *semantic heterogeneity*—also a main concern when dealing with structured data models—is exacerbated by the increased modeling flexibility of XML. Therefore, some assumptions about the semantics and arrangement of data items are necessary. These assumption are not too strong and, in fact, many real-world heterogeneous XML datasets are already in suitable format for our similarity join algorithms [LM09]. When this is not the case, data transformations have to be applied (see Section 2.1.2) to meet the assumptions described next.

Common Vocabulary

We assume that element labels are drawn from a common vocabulary. We do not address semantic matching of element nodes: labels are compared under the exact matching paradigm. In this work, we focus on similarity matching of structure and textual content. Operations used to support vocabulary integration such as schema matching and schema mapping are processes as complex as EM itself; it would be impractical to incorporate these operations into our similarity join framework, especially at the scale we have in mind. Note, nonetheless, that XML schema documents can be very large, in particular, those based on the W3C XML Schema specification [xml09]. Such schemas, very often designed for data validation, can be larger than the data instances they describe [RDM04]. In this context, similarity join techniques presented in this thesis can be used to support schema matching.

Tree-structured Entity Descriptions

We assume that all information needed to identify the entity of interest for matching, i.e., the entitiy description, is contained in well-defined trees. In other words, to compare an XML tree rooted at u with any other tree, we only consider node u and its descendants; elements preceding or following u in document order—or, equivalently, in preorder traversal—are not considered. This means that we only address matching of entities represented by tree structures. For example, consider Figure 2.4. We assume that all information describing each entity *exam* is contained in the node `exam` and the nodes contained in it (i.e., `patient`, `study`, `name`, "Bob", etc.). When entity description information is scattered in an XML document forming a (possibly disconnected) graph-like structure, structural transformations have to be applied to rearrange it in a tree-structured representation before our similarity join algorithms take place.

Entity descriptions can be represented by entire XML documents or document fragments. For the latter, we assume mechanisms to identify and fetch the subtrees of interest from an XML document. In Chapter 6, we will discuss such mechanisms in the context of XTC. Further, we assume that the subtrees corresponding to document fragments are identified by a common root node label, for example, the nodes with label `exam` in Figure 2.4. In case of recursivity, i.e., nested occurrences of the root node label, we consider only the topmost occurrence.

2.2.3 Problem Definition

A general tree similarity join takes as input two collections of XML documents (or document fragments) and outputs a sequence of all pairs of trees from the two collections that have similarity greater than a given threshold. The notion of similarity between trees is numerically assessed by a similarity function used as join predicate and applied to the specified node subsets of the respective trees.

Definition 2.1 (General Tree Similarity Join)**.** *Let C_1 and C_2 be two collections of XML trees. Given two trees T_1 and T_2, we denote by $sf(T_1, T_2)$ a similarity function on node sets of T_1 and T_2, respectively. Finally, let τ be a constant threshold. A tree similarity join (TSJ) between C_1 and C_2 returns all pairs $(T_1, T_2) \in C_1 \times C_2$ such that $sf(T_1, T_2) \geq \tau$.*

Note that the similarity function is applied to node sets instead of trees. When comparing trees, we need the flexibility to evaluate their similarity using node subsets that do not have containment relationships among them, e.g., node sets consisting only of text nodes. When structure matters, the function $rel(u, v)$ allows identifying containment (and sibling) relationships between a pair of nodes.

2.2.4 Related Approaches

Our XML data model is very similar to the models adopted by Guha et al. [GJK+06] and Augsten et al. [ABDG08]. Further, as noted before, similarity assessment on tree-structured data conveys the notion of co-occurrence similarity: if two entities contain similar subentities, then their similarity is increased. The main difference between our approach and that of Ananthakrishna et al. [ACG02] (also Weis and Naumann for XML data [WN05]) is that we do not assume structural homogeneity. Instead, we incorporate the structural similarity in the overall similarity assessment. Finally, notice that our model is very different (conceptually) from the collective matching model used, for example, in [PMM+02] (see discussion in Section 2.1.3): we perform pairwise matching and we only consider related entities that are involved in containment relationships, e.g., the similarity between two entities is not affected by the similarity of their siblings. However, we emphasize that, similar to the examples presented in 2.1.4, our similarity join framework can still be used to support collective matching approaches. For example, our similarity joins can be used to construct the initial similarity graph from which probability distributions can be derived.

2.3 Similarity Functions

The concept of similarity is closely related to the concept of duplication under the reasonable and widely used assumption that duplicates are likely to be similar to each other. The notion of similarity is captured by similarity functions. Informally, similarity functions provide a numerical measure for the degree to which two entities are ″alike“ or ″close“. Similarity values are usually given in the interval $[0, 1]$, where 0 denotes ″no similarity″ and 1 denotes “complete similarity“ or equality. The dual notion is *distance* that quantifies the degree to which two entities are ″different“ or ″far away“. In this thesis, we loosely use the term ″similarity“ to refer to both similarity and distance concepts. Note that, generally, similarity and distance values can be easily converted into one another by applying a simple expression of the form $similarity = 1 - distance$.

Very frequently, the similarity between two entities is not apparent from their original representation and operations have to be carried out to disclose it. Similarity assessment typically involves three components: a) the selection of relevant information from the given representation of an entity; b) the organization of this information in a new representation, which we call *entity description*; c) the manipulation of two entity descriptions to yield their similarity value. Component a) specifies in what respects the similarity between two objects are evaluated. This is a crucial aspect because the concept of similarity is incomplete in nature: it needs a frame of reference [MRLG93]. For example, two XML trees can be deemed as far apart or identical depending on whether we consider the sibling order relevant or not. Component b) provides a representation for the selected information that is suitable for processing by component c). This division is not crisp, however. For instance, component c) can directly operate on the original entity representation and implicitly select the relevant information for similarity. Also, note that entity description corresponds the set of comparison fields, which we discussed earlier in the context of EM on relational data.

Similarity functions can be associated to any (combination of) of the components described above. For example, the main aspect of a similarity function can be the technique used to select and generate the entitiy description or the algorithm employed to produce the similarity value. Further, as already noted in Section 2.1.2, data transformations are usually an integral part of similarity functions: they are implied in the generation of the entitiy description as well as can be embodied in the calculation of the similarity value (some transformations can be applied to the original entity representation before using a similarity function, too). Moreover, as we will see shortly, the very notion of similarity can be intrinsically connected to data transformation.

Despite the considerations above, it is not our objective here to provide a detailed analysis of the concept of similarity. We refer the interested reader to [Ric08] for thor-

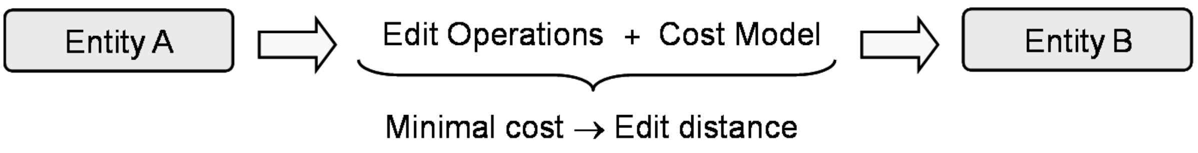


Figure 2.5: The concept of edit distance

ough discussion of similarity in the context of several application areas. We also do not aim to identify the best similarity function for the EM task. The choice of an appropriate similarity function is dependent on the purpose for which similarity is evaluated. In our case, the intended task is the identification of duplicates, which is, again, context-sensitive. Moreover, previous efforts in identifying a ”silver bullet“ similarity function were unsuccessful [ZM98, CRF03]: no similarity function was found to perform consistently well across all experimental domains. Therefore, we focus instead on two classes of similarity functions that have been extensively used in EM applications, namely edit-distance and token-based functions. In the following, we describe these two classes; we defer most of the considerations about effectiveness and efficiency as well other criteria to Section 2.3.3, where we analyze and compare edit-distance and token-based similarity functions in the light of our XML similarity join framework.

2.3.1 Edit-distance Similarity Functions

The similarity between two entities can be defined in terms of the ”amount of work“ to make them equal. This is the intuition behind the class of edit-distance similarity functions. As illustrated in Figure 2.5, given a set of editing operations and a cost model, the *edit distance* is obtained by the cheapest sequence of editing operations that transforms one entity into the another. Editing operations are typically applied to the original representation of entities, implicitly determining the information that is considered for similarity evaluation. The cost model defines costs depending on the editing operation or on the type of information involved. Generally, edit-distance formulations define a *metric*, i.e., it satisfies the properties of *symmetry*, *reflexivity*, *strict positiveness*, and *triangle inequality* [CNBYM01].

The *string edit distance* (SED) measures the distance between strings [Nav01]. In the simplest form, SED operations are restricted to character insertion, character deletion, and character substitution, and all operations have unit cost (equivalent to calculating the number of editing operations); this formulation is popularly known as the Levenshtein distance [Lev65]. Many variants have been proposed using different sets of editing operations and cost models, e.g., variants allowing character transposition, movement of blocks of contiguous characters, and cost assigning based on character position [GIJ+01, EIV07].

The classic algorithm for the Levenshtein distance is based on dynamic programming. There have been improvements to this basic algorithm in terms of runtime and space complexity. The fastest algorithms in practice employ a filtering strategy, some of them using token-based methods that we discuss in the next section [Nav01]. Nevertheless, dynamic programming algorithms, in general, have the merit of providing a flexible framework to accommodate different sest of operations and cost models and the classic algorithm is useful as an illustrative example for our discussion. Thus, we review it here.

Given two string s_1 and s_2, we first construct a matrix $M_{0\ldots|s_1|,0\ldots|s_2|}$. Then, the matrix is filled recursively as follows:

$$\begin{aligned}
M_{i,0} &= i \\
M_{0,j} &= j \\
M_{i,j} &= \text{if } \ (x_i = y_i) \ \text{ then } \ M_{i-1,j-1} \\
& \text{else } \ 1 + min(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1})
\end{aligned}$$

At each iteration, $M_{i,j}$ stores the minimal number of operations to transform $s_1[1, ..., i]$ into $s_2[1, ..., j]$; therefore, at the end of the algorithm, the distance between s_1 and s_2 is given in $M_{|s_1|,|s_2|}$.

The generalization of SED for tree-structured data is the *tree edit distance* (TED). Tai [Tai79] introduced the basic TED formulation for ordered trees — operations constrained to node insertion, node deletion, and node relabeling, all operations under a unit cost model — and provided a polynomial time algorithm. TED is inherently harder than SED, because node relationships, e.g., ancestor-descendant and sibling ordering, imply more data dependency thereby restricting the reuse of solutions to subproblems in dynamic programming algorithms. For example, in the SED algorithm above, if the characters at position i and j are equal, then we always have $M_{i,j} = M_{i-1,j-1}$. For TED, node relationships may prevent an analogous implication from holding [ZS89]. Nonetheless, improvements on Tai's algorithm regarding runtime complexity have appeared in [ZS89, Kle98, DMRW07]; all these algorithms can be categorized in terms of the *cover strategy framework* defined in [DT03].

As for SED, there is a plethora of variants of TED employing different sets of editing operations and cost models, e.g., allowing insertion and deletion on leave nodes only and subtree movements and operation costs proportional to the fanout of the involved nodes. In Chapter 3, we analyze several TED variants as well as approximations thereto employing token-based methods.

Of course, besides the set of edit operations and cost model, other aspects can be embedded in the algorithmic details of edit-distance functions. For example, the version of TED for unordered trees is actually an algorithm different from that for ordered trees, that can be defined over the same edit operations and cost model. Nevertheless, despite of assuming a common algorithmic framework, the abstraction of edit distances depicted in Figure 2.5 is still useful to guide our discussion, in partic-

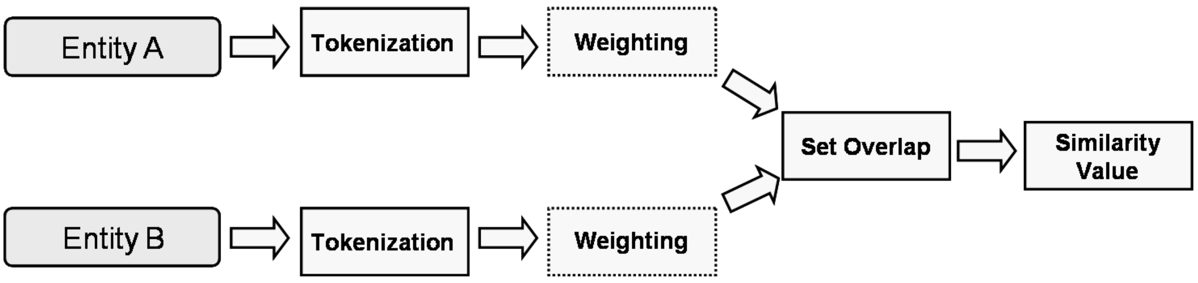


Figure 2.6: Token-based similarity functions

ular, for the comparison between edit-distance and token-based similarity functions presented in Section 2.3.3.

We now formally define the general edit distance (GED) and the *general edit similarity*, a similarity function defined in terms of GED.

Definition 2.2 (General Edit Similarity). *Given two entities e_1 and e_2, the edit distance $ED(e_1, e_2)$ between them is the minimal cost of a sequence of edit operations that transforms e_1 into e_2. There is a finite number of edit operations; each operation is of the form $ed(\sigma 1, \sigma 2) = c$, where σ_1 and σ_2 are results of selections of units of information over an entity (e.g., characters, nodes, and subtrees) or the null symbol ϵ, and c is a non-negative real number defining the cost of the operation. The cost of a sequence is the sum of the costs of the individual operations. Let $|e|$ be the size of an entity according to its smalles unit of information under consideration (e.g., given by the number of characters in a string or nodes in a tree). The general edit similarity between e_1 and e_2 is defined as follows.*

$$GES(e_1, e_2) = 1 - \frac{ED(e_1, e_2)}{max(|e_1|, |e_2|)} \tag{2.6}$$

2.3.2 Token-based Similarity Functions

Many similarity functions unfold two basic operations: *a)* transformation of entities of interest into sets containing the smallest units of information, which we call henceforth *tokens*, and, afterwards, *b)* assessment of their similarity based on the number of tokens they have in common; additionally, prior to *b)*, *c)* weights can be associated with tokens to quantify their relative importance. We refer to operation *a)* as *tokenization*, *b)* as *set-overlap measurement*, *c)* as *weighting*, and similarity functions employing these operations as token-based similarity functions[11]. Figure 2.6 depicts the course

[11] Alternatively, such functions can be referred to as *set-overlap-based similarity functions* [RH08b]. The choice is based on which operation is emphasized: tokenization or set-overlap measurement. Therefore, here and in Chapters 3 and 4, we shall use the term token-based. In Chapter 5, we switch to set-overlap-based, as we focus on the set-overlap measurement operation.

of a token-based similarity function along its three components towards a similarity value. We discuss each of these components in the following.

Tokenization

Tokenization is essentially a method for selecting and organizing relevant pieces of information from entities. It can be applied to the content part of an XML document, its structural part, or both, to capture information such as patterns in strings and node relationships. The key idea behind tokenization methods is that most of the tokens derived from significantly similar entities should agree accordingly. As a result, the token sets of two similar entities would only have a large overlap and, in turn, a high similarity value.

Tokenization is carried out by a *tokenization function* which splits an entity into a set of tokens. We denote by $tok[\rho_1, \dots, \rho_n]$ a tokenization function, where $\rho_1, \dots, \rho_n$ are free parameters; given an entity e, we refer to the set of tokens $tok[\rho_1, \dots, \rho_n](e)$ as the $tok[\rho_1, \dots, \rho_n]$ *profile* of e. Whenever clear from the context or unimportant for the discussion, we will omit the list of parameters of a tokenization function, i.e., referring to the corresponding set of tokens as the *tok* profile of a entity e, or even omit the tokenization function altogether (e.g, simply saying the *profile* of e). In the latter case, we denote by $\mathcal{P}$ an arbitrary profile.

Further, we denote by $t_1 \circ t_2$ the concatenation of two arbitrary tokens t_1 and t_2. The symbol $\circ$ denotes a special pattern that cannot be derived from an entity by any tokenization or transformation method. Thus, we have $t_1 \circ t_2 = t_3$ iff t_3 represents the concatenation of tokens t'_1 and t'_2 and $t_1 = t'_1$, $t_2 = t'_2$.

We now discuss tokenization functions for content and structure with emphasis on the former. Several structural tokenization functions are presented and evaluated in Chapter 3. The generation of tokens that jointly capture textual and structural information is introduced in Chapter 4.

The concept of q-grams is a well-known approach to tokenize strings [Ukk92]. Informally, a q-gram is a substring of size q. We denote by $qgram[q]$ the tokenization function that maps a string s to the profile $qgram[q](s)$. There are several ways of splitting a string into a set of q-grams — hence, several variants of the $qgram$ function exist; many of these techniques are designed for filtering-based SED algorithms [Nav01]. For token-based methods, the common procedure consists of "sliding" a window of size q over the characters of s. It results in the following profile cardinality: $|qgram[q](s)| = |s| - q + 1$.

Example 2.7. *Consider the strings* $s_1 = Kaiserslautern$ *and* $s_2 = Kaserslatern$*. Their* $qgram[3]$ *profiles profiles are:*

$qgram[3](s_1) =$ {'Kai', 'ais', 'ise', 'ser', 'ers', 'rsl', 'sla', 'lau', 'aut', 'ute', 'ter', 'ern'},

$qgram[3](s_2) =$ {'Kas', 'ase', 'ser', 'ers', 'rsl', 'sla', 'lat', 'ate', 'ter', 'ern'}.

The value of q presents a trade-off between efficiency and effectiveness. Higher values of q are likely to result in less frequent grams for a dataset — the frequency

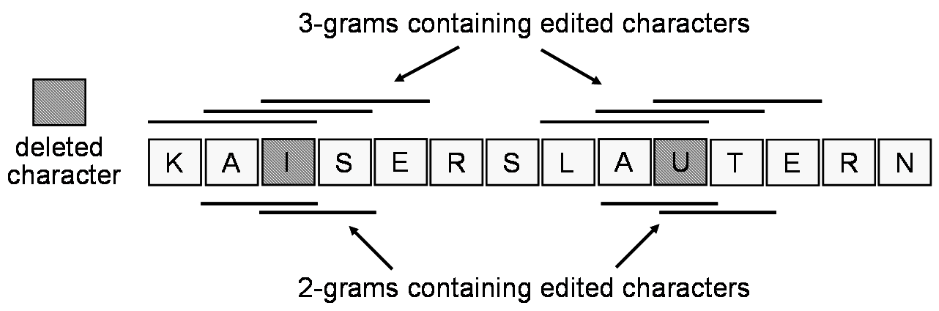

3-grams containing edited characters
deleted character
K
A
I
S
E
R
S
L
A
U
T
E
R
N
2-grams containing edited characters

Figure 2.7: q-grams affected by editing operations, q-2 and q=3

of a string cannot be higher than that of any of its substrings. Intuitively, it leads to fewer strings having a q-gram in common and, thus, more comparisons can be safely avoided (because empty profile overlap means ”no similarity“). In Chapter 5, we describe methods that exploit token frequency, token set size, and the threshold value to reduce even more the number of set-overlap calculations. Unfortunately, the accuracy normally drops as the value of q increases. To see why, observe that the number of q-grams ”destroyed“ in $qgram[q](s)$ due to edit operations on s is related to the value of q: $O(kq)$ q-grams are destroyed after k edit operations [Ukk92]. Figure 2.7 illustrates this aspect for $q = 2$ and $q = 3$. Thus, higher values for q penalize more character mismatches; as q increases, q-gram-based similarity gets closer to equality comparisons. It has been empirically observed that $q = 2$ and $q = 3$ give the best accuracy results [CHK$^+$07]. Note that for $q = 1$, different strings containing identical (multi-) sets of characters have the same $qgram$ profile and, hence, maximum similarity. For $q > 1$, more complex patterns are required to get this (unwanted) effect [Ukk92][12].

It is common to (conceptually) extend a string s by prefixing and suffixing it with $q-1$ null symbols. Therefore, all characters in s participate in exact q q-grams and mismatches at any position of s are uniformly weighted. In addition, all white spaces between two words can be replaced by $q-1$ null symbols [CHK$^+$07]. As a result, q-gram similarity is made fully insensitive to word ordering, e.g., ”Leonardo Ribeiro“ $=$ ”Ribeiro Leonardo“. Again, such approaches trade efficiency for accuracy. Additional characters result in larger sets and therefore more processing overhead.

Structural tokenization functions operate on element nodes. The most basic function consists of collecting all element node labels of a tree in a bag of labels, thus ignoring all the tree structure. We call this tokenization function $labels$. More elaborate methods exploit parent-child, ancestor-descendant, and sibling relationships. We give an example of a tokenization function based on the parent-child relationship in the following.

[12]Note that, therefore, token-based similarity functions define a pseudo-metric, because *strict positiveness* does not hold.

Example 2.8. *Consider a structural tokenization function* $parchild$ *which collects all pairs of element nodes involved in parent-child relationships. The* $parchild$ *profiles of the subtrees* a) *and* b) *shown in Figure 2.4 are:*

$parchild(a)$ ={`exame` ∘ `patient`, `patient` ∘ `study`, `study` ∘ `id`, `study` ∘ `description`, `patient` ∘ `name`, `patient` ∘ `relatives`, `relatives` ∘ `mother`},

$parchild(b)$ ={`exame`∘`study`, `study`∘`patient`, `patient`∘`name`, `patient`∘`mother`, `study`∘ `id`, `study` ∘ `description`}.

Weighting Schemes

In many domains, tokens show non-uniformity regarding some semantic properties, such as discriminating power. In fact, previous work consistently revealed that not all tokens are equally important for similarity evaluation [SM83, Jon72, CHK+07]. The process of quantifying this importance is referred to as *weighting* and the method employed for assigning weights to tokens is called *weighting scheme*. In other words, a weighting scheme ws converts a profile $\mathcal{P} = \{t_0, \ldots, t_n\}$ into a weighted profile $ws(\mathcal{P}) = \{\langle t_0, w(t_0)\rangle, \ldots, \langle w(t_n)\rangle\}$, where $w(t_0)$ is the associated weight of token t_i.

Common wisdom from the IR field dictates that tokens appearing very frequently in a collection of documents contribute to the discrimination of them to a lesser degree, whereas rare tokens usually carry more content information and are more discriminative. Such observation is in accordance with the frequency-based approximation of Newcombe et al. in the EM context, as observed before. The *inverse document frequency* (*IDF*) weighting scheme is popularly used to capture this intuition. The *idf* weight of a token t is inversely proportional to the total number of trees $freq(t, C)$, in which t appears in a collection C.[13] A typical *idf* formulation is given by:

$$idf(t) = ln(1 + \frac{|C|}{freq(t, C)}) \ . \tag{2.7}$$

The *term frequency* (*TF*), i.e., the frequency of a token in a tree, is also used for weighting. Given a token t appearing $freq(t, T)$ times in a tree T, its *tf* weight can be computed as follows:

$$tf(t) = 1 + ln(freq(t, T)) \ . \tag{2.8}$$

The product of both term statistics constitutes the well-known *TF-IDF* weighting scheme. Several other weighting schemes have been successfully used in IR. In particular, we mention the Okapi BM25 [RW94], which is based on probabilistic retrieval models. An adaptation of Okapi BM25 to XML data is presented in [TSW05].

The utility of the *tf* weight for string-to-string similarity comparison has been questioned [HCKS08]. In IR, a user-formulated query is compared to normally much larger documents and it is reasonable to consider the frequency of a query token in a

[13]Note that we have overloaded $freq(t, C)$. In Section 2.1.1, $freq(v, C)$ denotes the frequency of value in a collection C, whereas, here, $freq(t, C)$ denotes the number of trees in which a token appears, irrespective of how many times it occurs in each tree.

document as an indication that the document is relevant to the query. However for string-to-string comparison, the intuition is just the opposite: frequency discrepancy of the same token present in two strings should decrease their similarity. Moreover, EM systems usually perform comparisons on much shorter strings, e.g., author name fields. A simple solution consists of converting a profile $\mathcal{P}$, composed by a bag of tokens, into a profile $\mathcal{P}_a$, composed by a set of *annotated tokens*, by concatenating the symbol of a sequential ordinal number to each occurrence of a token. For example, profile $\mathcal{P} = \{a, b, b\}$ is converted to $\mathcal{P}_a = \{a \circ 1, b \circ 1, b \circ 2\}$. For annotated tokens, we have $\mathit{freq}(t, T) = 1, \forall t$; thus, their *tf-idf* weights are determined by the idf weight only. Furthermore, divergence in term frequencies of a token will penalize the similarity of the related strings by a stronger degree, because subsequent occurrences of the referenced token have decreased the document frequency in a collection (and, therefore, increased the *idf* weight). Unless stated otherwise, we use annotated profiles (without indication in the subscripts).

Set-overlap Measurement

Tokenization delivers an XML tree represented as a set of tokens. Afterwards, similarity assessment can be reduced to the problem of set overlap, where different ways to measure the overlap raise various notions of similarity. Fortunately, there are several set-overlap similarity measures available; most of them can be rewritten into equivalent set-overlap predicates [SK04, CGK06]. Next, we formally define the Jaccard, a widely used set-overlap similarity measure and its version for weighted profiles. We present the corresponding set-overlap predicate for Jaccard as well as for other measures in Chapter 5.

Definition 2.3 (Jaccard Similarity (JS))**.** *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two profiles. The Jaccard similarity (JS) of $\mathcal{P}_1$ and $\mathcal{P}_2$ is defined as follows:*

$$JS(\mathcal{P}_1, \mathcal{P}_2) = \frac{|\mathcal{P}1 \cap \mathcal{P}2|}{|\mathcal{P}_1| + |\mathcal{P}_2| - |\mathcal{P}_1 \cap \mathcal{P}_2|} \, . \tag{2.9}$$

Definition 2.4 (Weighted Jaccard similarity (WJS))**.** *Let $ws(\mathcal{P}_1)$ be a weighted profile and $w(t, ws(\mathcal{P}_1))$ be the weight of a token t in $ws(\mathcal{P}_1)$. Let the weight of $ws(\mathcal{P}_1)$ be given by $w(ws(\mathcal{P}_1)) = \sum_{t \in ws(\mathcal{P}_1)} w(t, ws(\mathcal{P}_1))$. Similarly, consider a profile $ws(\mathcal{P}_2)$. The weighted Jaccard similarity (WJS) of $ws(\mathcal{P}_1)$ and $ws(\mathcal{P}_2)$ is defined as follows:*

$$WJS(ws(\mathcal{P}_1), ws(\mathcal{P}_2)) = \frac{w(ws(\mathcal{P}_1) \cap ws(\mathcal{P}_2))}{w(ws(\mathcal{P}_1)) + w(ws(\mathcal{P}_2)) - w(ws(\mathcal{P}_1) \cap ws(\mathcal{P}_2))} \, , \tag{2.10}$$

where the set overlap of $ws(\mathcal{P}_1)$ and $ws(\mathcal{P}_2)$ is given by:

$$w(ws(\mathcal{P}_1)) \cap ws(\mathcal{P}_2) = \sum_{t \in ws(\mathcal{P}_1) \cap ws(\mathcal{P}_2)} min(w(t, ws(\mathcal{P}_1)), w(t, ws(\mathcal{P}_2))) \, . \tag{2.11}$$

In the definition above, *minimum* (min) is used to aggregate varying token weights across different profiles. This situation happens, for example, when the weights are normalized like in [HCKS08]. Another example is the weighting scheme presented in Section 3.2.1.

Example 2.9. *Consider the strings* $s_1 = Kaiserslautern$ *and* $s_2 = Kaserslutern$ *and their respective qgram profiles* $qgram[3](s_1)$ *and* $qgram[3](s_2)$ *shown in Example 2.7. Therefore, we have:*

$$JS(s_1, s_2) = \frac{6}{12 + 10 - 6} = 0.375 \; .$$

2.3.3 Edit-distance vs. Token-based

We now compare edit-distance and token-based similarity functions in the context of XML similarity joins. We identify four main criteria for comparison: *effectiveness*, *efficiency*, *versatility*, and *text and structure combination*, which we shall discuss in this order. Note that these criteria are mostly interrelated. Hence, despite discussing each criterion separately, we make some cross-references when necessary. Finally, we present the summary of our comparison.

Effectiveness

Effectiveness is the ability of a similarity measure to capture a given notion of similarity thereby determining the quality of the results reported. This is the most important criterion. Our discussion is based on reported results from previous work; details about the metrics and assumptions used in our own effectiveness measurements are deferred to the next section.

By far and large, edit-distance functions have been explored under the unit-cost model and using a basic set of editing operations — insertion, deletion, substitution (or relabeling for trees), and transposition. For textual similarity, SED performs very well for capturing typographical errors such as misspellings. However, it is typically ineffective for other kinds of mismatches, especially those due to word transpositions. Hence, it is problematic for non-standardized or unstructured text. TED may also exhibit poor performance on some data. For example, insertion and deletion operations change the tree structure, which have side effects on other nodes. This may lead to non-intuitive results [ABG10]. Many of these shortcomings can be mitigated by extending the set of operations allowed or employing non-uniform cost models.

In general, token-based functions have been shown to provide competitive accuracy results with edit-distance methods [CRF03, CHK+07, ABG10]. Further, weighting schemes can be used to regard the relative token importance for similarity and improve the quality. A drawback of token-based methods is that, sometimes, different entities may yield the same token set leading to an erroneous maximum similarity result.

Efficiency

Efficiency refers to the algorithmic complexity of the similarity function in terms of space and execution time. In addition, it is also important how the similarity function lends itself to optimizations such as derivation of bounds, partitioning, or indexing.

The best runtime complexity for SED is $O(kn)$ for k errors and strings of size n [Nav01]. For ordered trees, the current best result is in $O(n^3)$ for trees with n nodes [DMRW07]; algorithmic improvements can be obtained by restricting the set of edit operations or sequence of operations that can be applied to a tree (e.g., [NJ02]). For unordered trees, TED has been to shown to be NP-hard [ZSS92]; a heuristic solution in $O(n^3)$ is presented in [CGM97].

Token-based methods are computationally much more efficient. Tokenization and weighting can be carried out in a single pass over the string or tree. The dominant time cost pertains to overlap measurement, whose runtime complexity is $O(nlogn)$ using sorting or even $O(n)$ if hash structures are used. The space complexity of token-based functions is $O(n)$. Because of its time and space efficiency, token-based methods are frequently used to approximate edit distances. For example, token-based approximations for TED on ordered and unordered trees are presented in [ABG10] and [ABDG08], respectively.

When dealing with large datasets, minimizing the number of pairwise similarity computations is more important than efficiently computing the similarity function. A common method for pruning candidates in edit-distance joins is based on token-based methods [GIJ$^+$01, YKT05], as mentioned before. Another popular approach consists of exploiting the metric properties of edit distances to map entities from the original metric space into a vector space; afterwards, multi-dimensional access methods (e.g., R-Trees) can be used to save distance calculations [JLM03, GJK$^+$06]. However, note that such methods require the pivots being selected from the same universe, i.e., from the combination of all source datasets. Clearly, it might be impractical in Web-scale scenarios such as dataspace systems.

For token-based methods, index structures such as B-trees and inverted lists are commonly employed to efficiently find all sets that share at least one token; irrelevant pairs (with empty token set overlap) are never considered. Furthermore, there is a wealth of techniques for aggressively pruning the comparison space including: signature schemes [AGK06], exploitation of token and set collection ordering [SK04, CGK06, BMS07], pruning using overlap and size bounds [XWLY08, SK04, AGK06], dimension reduction [Bro97] methods, and index reduction techniques [BMS07, RH09].

Versatility

It is well-known that no single similarity function is the best for all applications and scenarios (e.g., see [CRF03] for string similarity functions). Hence, it is very desirable to have a rich similarity space, in which different notions of similarity can be easily obtained by changing simple parameters.

Edit distance methods deliver different notions of similarity by varying the set of edit operations allowed (e.g., transposition of characters, subtree deletion and insertion) or by changing the cost model (e.g., cost of changing 0 to O might be smaller or the cost for changing nodes at lower nesting levels might be larger). While it is relatively easy to accommodate different editing operations and cost models for SED onto a common (dynamic programming-based) foundational component [Nav01], the same cannot be assumed for TED. For example, the version presented in [NJ02], which allows subtree operations, considerably differs in algorithmic details from Zhang and Shasha's version [ZS89] only allowing node-level editions. Thus, identifying a generic abstraction between such algorithms to uniformly and flexibly support different editing operations and cost models is difficult.

The three components of token-based functions — tokenization, weighting, and overlap measurement — are fully independent. Thus, they can be freely combined to obtain different notions of similarity. Moreover, their computation can be performed in a pipelined fashion, and therefore it can be conveniently performed within database query engines.

Text and Structure Combination

Specific for tree-structured data, this criterion refers to the ability of a similarity function to evaluate textual and structural similarity in a unified way. This aspect is crucial for performing a combination of evidence smoothly and efficiently.

For both classes of similarity functions, it is possible to employ the score-based or rank-based combination approaches (see Section 2.1.2). Guha et al. [GJK+06] proposed incorporating SED inside the TED algorithm as an additional function call if the data is of string type. In this context, it is conceivable to devise a cost model incorporating both structural and textual operations. Of course, this approach implies increased time complexity to the already expensive TED computation.

For token-based approaches, text and structure combination can be performed at the token level without imposing significant overhead owing to separate subroutines or operators. The tokenization operation can produce profiles containing structural and textual tokens and weighting schemes can be employed to account for the relative token importance; similar approaches have been used in XML retrieval [CMM+03]. In this connection, we can produce tokens that jointly capture structural and textual information and, thus, combine both aspects of similarity into a single measure.

Comparison Summary

Token-based and edit-distance similarity functions have been shown to provide comparable effectiveness, the most important criterion. Regarding efficiency, token-based functions are overwhelmingly superior. In fact, token-based approaches are widely used as filters in distance joins that employ edit-distance functions in the similarity predicate. More importantly, token-based methods allow much more opportunities for saving similarity calculations. Further, token-based functions provide more flexibility to accommodate several notions of similarity in a common framework as well as to combine evidence from textual and structural similarity. Therefore, the class of token-based similarity functions is our method of choice for designing effective, efficient, and versatile XML similarity join algorithms and shall be our focus henceforth in this thesis.

2.3.4 Token-based Similarity Function Notation

A token-based similarity function is defined by the triple $\langle tok, ws, ss \rangle$, where tok is a tokenization function, ws is a weighting scheme, and ss is a set-overlap similarity measure. Further, we can define *classes* of token-based similarity functions by letting one or two elements of the triple unspecified. For example, $\langle tok, \_, \_ \rangle$ defines the class of token-based similarity functions using tok together with any weighting scheme and set-overlap measure. The weighting scheme is the only element that can be absent, i.e., $ws = \epsilon$. Of course, ws and ss are partially correlated: when a weighting scheme is given, ss must be a set-overlap similarity measure for weighted sets; otherwise for unweighted sets.

We focus on the tokenization function for most of the token-based similarity functions presented in Chapters 3 and 4, i.e., we focus on the class defined by $\langle tok, \_, \_ \rangle$. Moreover, in such cases, we denote by TOK a class of similarity functions using tok as tokenization function (the name of tokenization function in uppercase); the corresponding similarity value between trees T_1 and T_2 is therefore given by $TOK(T_1, T_2)$.

2.4 Quality Measurements

We now describe in detail the methods used in this thesis for measuring the effectiveness of similarity functions and, in turn, similarity joins in identifying duplicates. Quantifying the quality of similarity function results involves several subjective decisions. For example, what properties of the results are important to measure? What are the assumptions behind the choice of a given property and a corresponding measure? Therefore, before presenting evaluation metrics that objectively quantify some desired property, we will justify the choice of this property. This section is dedicated to effectiveness measurements. Efficiency measurements considered in this thesis are based on more straightforward and inherently quantifiable properties, such as runtime performance, and will be addressed in Chapter 5.

2.4.1 Experimental Approach

First and foremost, we have to consider what are the *utilities* of similarity functions in the EM context. In this vein, an obvious utility is to support (correct) matching decisions. Another closely related aspect is the amount of human effort required. The task of identification of duplicates is intrinsically semi-automatic. Therefore, besides support of automatic decisions, it is also important that similarity function results contribute to maximize the efficiency of the activities requiring human interaction.

Following the terminology used in [ZB06], we refer to the assumption allowing a particular kind of evidence to be used to support a hypothesis of utility as *warrant*.

Similarity functions are employed in similarity join predicates together with a constant threshold. Hence, an intuitive warrant is that a similarity function is useful to support matching decisions when it yields higher similarity values for pairs of duplicates than for pairs of non-duplicates: if so, one can specify a threshold to separate duplicates from non-duplicates. We will evaluate similarity functions in isolation as well as in combining approaches. For the first case, while it has been shown that the most successful combinations do not necessarily include the best set of similarity functions — for example, the independence between similarity functions is an important criterion [PC98, OC03] — , it is reasonable to assume that the best performing similarity function will be nevertheless present in any combining formulation.

We will use standard IR evaluation measures [SM83]. Moreover, despite our focus on similarity joins, we will report experiments conducted in a similarity selection setting: we consider a tree from a dataset as a query and the number of true duplicates of this tree as the set of relevant answers. Similarity selection can be viewed as the special case of similarity join where one of the join partners has only one entry. We did not observe any significant deviation from the result obtained in a similarity join setting (e.g., [BM03b]); all results followed identical trends. We use a similarity selection setting for the following reasons. First, it is more straightforward to use IR evaluation measures. Although evaluating similarity join results is useful to reveal the best threshold value in terms of accuracy, the selection of thresholds is not our focus in this thesis. Second, and more importantly, we view the order of the results, i.e., the ranking, as a very important aspect in an EM scenario. In particular, the relative ordering between duplicates and non-duplicates has influence on the user interaction and performance as we will discuss shortly. Some evaluation measures for ranked results are more easily calculated using similarity selection. Indeed, we will not use any threshold parameter in our experiments and, therefore, the rank returned is *complete*.

To evaluate the accuracy of similarity functions, we need *golden-truth* datasets, i.e., datasets in which all duplicates are identified. We will use publicly available real-world XML datasets (e.g., DBLP [dbl09]) as source datasets. Golden-truth datasets are derived by generating data containing artificially generated duplicates. To this end, we produced duplicates by performing transformations on content and structure of XML subtrees. We kept track of all duplicates generated from each subtree; those duplicates form a partition and carry the same identifier called *duplicate ID*. Details about the datasets and the methods for duplicate generation used for the experimental evaluation are presented in the upcoming chapters.

2.4.2 Evaluation Measures

We are now ready to present the evaluation measures. For all measures, we will compute the mean over a set of queries. Unless explicitly stated otherwise, we use a query workload of 100 queries.

First, let us provide some notation. Given a collection of XML trees C, let $T \in C$ be a *query tree*, an XML tree used as similarity selection predicate, and $R_T \subseteq C$ be the set of trees returned in the result. Accordingly, let $D_T \subseteq R_T$ be the subset of duplicates of T in the result and $N_T \subseteq R_T$ the subset of non-duplicates. We will also use other subsets of R_T. Let $R_{p|T} \subseteq R_T$ generally represent the subset of R_T satisfying the predicate p. Note that $R_{p|T}$ further induces the subsets $D_{p|T}$ and $C_{p|T}$.

Most evaluation measures presented here are build upon the basic concepts of *precision* and *recall*. We begin with measures that directly quantify these concepts.

The intuitive warrant of precision is that of *exactness* of the results. We define the precision of the result of a query tree t as follows:

$$Pr(T) = \frac{\mid D_T \mid}{\mid R_T \mid} \tag{2.12}$$

The warrant of recall is that of *completeness* of the results. We define the recall of the results of a query tree T as follows:

$$Re(T) = \frac{\mid D_T \mid}{\mid D \mid} \tag{2.13}$$

Precision and recall can then be plotted together in the so called *precision-recall graph*, which allows to observe precision results at different recall levels — we denote the precision at recall value re by $Pr(T, re)$. However, analyzing precision-recall graphs is problematic, because precision values are not exactly defined at a given recall level, i.e, we can have different precision values at a same recall level. The opposite is also true when precision is 1.0. Moreover, it is also difficult to compute the average of a set of queries results. For this reason, we calculate *interpolated precision values*: the interpolated precision at a recall level re is given by:

$$Pr_i(T, re) = \max_{re\prime \geq re} Pr(T, re') \tag{2.14}$$

In our experiments, we will report the *11-point interpolated average precision*, i.e, the average of the interpolated precision at recall levels 0.0, 0.1, ..., 0.9, 1.0.

It is also useful to analyze precision and recall values combined into a single expression. The standard way of combining precision and recall is by taking their harmonic mean, the so-called F_1 measure. To adapt F_1 for ranked results, we define $R_{T\prime|T}$ as the subset of R_T formed by T' and all other trees ranked before T'. Precision and recall for $R_{T\prime|T}$ are denoted by $Pr(T, T')$ and $Re(T, T')$, respectively. The maximum F_1 value over the elements in R_T, denoted as MF_1 is defined as follows:

$$MF_1(T) = \max_{t\prime \in R_T} \frac{2 \times Pr(T, T') \times Re(T, T')}{Pr(T, T') + Re(T, T')} \tag{2.15}$$

We consider the evaluation of the relative ranking positions of duplicates and non-duplicates. To this end, a popular measure is the average precision, denoted by $AP(t)$, which returns higher values when duplicates are in top positions. Besides automatic classification, we view support of human interaction as a warrant of this measure.

As we discussed earlier, in many EM scenarios, automatic classification is practically ruled out due to the high cost of misclassification. In the absence of more sophisticated supporting mechanisms for user feedback (see Section 2.1.3), the natural procedure is to select only the most similar candidate pairs for manual classification, i.e, candidates at top positions in the ranking. The average precision is defined as follows:

$$AP(T) = \frac{1}{\mid D_T \mid} \times \sum_{T\prime \in D_T} Pr(T, T') \tag{2.16}$$

In our experimental charts, we will report the mean of AP values over query workload as MAP.

The performance of a similarity function under all the above measures hinges to a variable extent to its ability of delivering higher similarity values for pairs of duplicates than for pairs of non-duplicates. We call this property of similarity functions *monotonicity*. We now define a novel measure that explicitly quantifies the monotonicity property as follows:

$$Mon(T) = \frac{1}{\mid D_T \mid \times \mid C_T \mid} \times \sum_{T' \in D} \mid C_T \setminus C_{T'|T} \mid \tag{2.17}$$

Similar to AP, the Mon measure returns the highest value when all duplicates are ranked before all non-duplicates. However, AP emphasizes ranking of duplicates at the very top positions. In contrast, Mon uniformly weights duplicate pairs that are ranked higher than non-duplicates, in whatever position. Besides quality of results, the monotonicity property has been exploited to formulate efficient query plans for EM [CCGK07][14].

Note that the measures above do not account for the presence of tied similarity values in the results. Tied scores may affect the accuracy of the evaluation measures, because they impose only a partial ordering on the result set; as a result, there are multiple possible orderings, each one leading to a different measurement result. Variants of several measures used here that regard the presence of ties are presented in [MN08]. We evaluated these variants in our experiments and observed identical trends. Therefore, we report only the results for original measures.

2.5 Summary

In this chapter, we presented background material for the upcoming chapters in this thesis. We provided an extensive overview of the EM problem. We started by discussing early work in this area and identifying its main influential ideas. From this

[14]The work in [CCGK07] defines the monotonicity property in terms of a set of similarity functions and in a binary fashion: it is required that any pair of duplicates has a higher similarity value than a non-duplicate on at least one similarity function.

discussion, we derived a general EM framework that we used as reference for reviewing modern EM approaches. More recent work representing a more marked departure from classical EM approaches was discussed as well. The importance of similarity joins in the EM context was highlighted in several examples. We then concentrated our discussion on XML similarity joins. We presented the XML data model and discussed some assumptions needed before providing the formal definition of the XML similarity join problem. Further, we discussed the most important classes of similarity functions for EM applications: edit-distance and token-based similarity functions. We compared these two classes and found token-based similarity functions superior according to relevant criteria for the context of our work. Finally, we presented our strategy and methods that will be used in this thesis for measuring the effectiveness of our algorithms.

Chapter 3

Similarity Functions for XML Structure

The structure of XML data conveys valuable information for distinguishing documents in a collection from each other. In the relational world, data is regular and homogeneous. The underlying structure has no discriminating power; indeed, structural information (or metadata) is stripped away from the data itself and stored in a separate catalog. XML, on the other hand, can be used to represent semi-structured data; this means that the data can be irregular, heterogeneous, and incomplete. In such cases, it is not possible to specify a fixed schema in advance. Moreover, description of the structure of the data is represented with the data itself, i.e., the data is self-describing. This latter aspect means that the distinction between data and structure is blurred in XML. As a result, very frequently, a considerable part of the information about an entity is embodied in its structure. Furthermore, from a semantic viewpoint, structure has special importance regardless of its discriminative power. For example, two XML documents having a completely different structure would certainly be classified as non-duplicates. Therefore, structural information represents an important feature of XML data that cannot be ignored by EM tasks.

As for strings, XML data representing the same entity can present slightly deviating structures. We view duplicate XML documents that exhibit structural discrepancy as *modeling variants* of each other. In the EM context, the aim of a structure-conscious similarity function is therefore to numerically quantify such variations. This information is then be used to compose the overall similarity between XML documents and, ultimately, classify them as duplicates or non-duplicates.

A related problem is that of identifying XML documents that have been generated by the same DTD [NJ02, FMM^{+}05, DCWS06, Hel07]. We refer to this problem as the *common DTD identification problem* (CDI problem, for short). In this regard, the main desideratum for similarity functions is the identification of mismatches in element definitions. Also, the similarity evaluation is expected to be less sensitive to divergences in the number of occurrences of subtrees, because they may be due to elements declared as optional or allowing multiple occurrences. In EM, on the other hand, while it is reasonable to assign lower similarity to XML trees that are likely to induce different DTD definitions, neglecting subtree occurrence mismatches can produce non-intuitive similarity results. For example, consider two trees T_1 and T_2 from a bibliographic database representing information about *articles*; T_1 contains one subelement `author` whereas T_2 contains five. Although T_1 and T_2 may well have been generated by the same DTD, the evidence that they are duplicates is weaker. Clearly, a similarity function that ascertains a high similarity to T_1 and T_2 would contradict the notion of co-occurrence similarity discussed in Chapter 2. We empirically evaluate this conjecture later in this chapter as we analyze the effectiveness of several approaches originally proposed for CDI in the EM context.

A more general problem is clustering XML documents by structure. Similarity functions employed to deal with this problem can be used in EM as well—as already mentioned, EM can be modeled as a clustering task that groups duplicate entities. In this context, Dalamagas et al. [DCWS06] applied TED on structural summaries of XML documents; a tokenization approach akin to the *parent-child* method illustrated in Chapter 2 is employed (on graph-represented XML trees) by Lian et al. [LCMY04]. Some work exploits frequent structural patterns to derive shorter tree representations (e.g., see [ATW+07]). This approach is similar in spirit to the feature selection techniques based on document frequency (see [YP97] and discussion in Chapter 2). A major drawback of such approaches is the high cost associated with the algorithms used to mine frequent structures [Zak02]. In contrast, we consider in this thesis inexpensive methods that allow, for example, on-the-fly tokenization of tree structure by similarity join sub-operators (see Chapter 6).

Several papers exploit structural similarity for change detection in tree-structured data [CRGMW96, CAM02]. Most approaches adopt a two-step strategy for similarity computation: first a mapping between the nodes of two trees is derived and then this mapping is used to generate a minimum-cost edit script. A fundamental difference between change detection and EM applications is that, in the former, comparisons are performed on trees from the same data source that typically exhibit little deviations. Such assumptions are too strong in the EM context where data sources can be disparate and contain highly heterogeneous trees. In this situation, the algorithms for tree similarity evaluation presented in [CRGMW96, CAM02] would be inefficient (see also discussion in [ABG10]).

The role of structure has been intensively explored in the context of XML retrieval. There exists empirical evidence that using structure in queries improves precision at low recall levels, but hurts accuracy at high recall levels [KMdRS06]. In EM, the number of relevant answers is expected to be substantially smaller in comparison to XML retrieval. In fact, because common real-world datasets contain small amounts of duplicates, most XML trees have either no duplicates or only a small number of them. Therefore, the fraction of relevant elements, for which the use of structure has been shown to enhance precision in XML retrieval (first retrieved elements), matches the typical recall range in the EM context.

This chapter is dedicated to the investigation of structural similarity functions in the EM context. The main contribution of this chapter is a novel method for producing compact and high-quality structural representations of unordered, document-centric XML trees (originally presented in [RHP09]). The representation we propose consists of structural tokens where each token embodies a set of similar structural patterns; such sets are identified in a pre-processing step. Therefore, at evaluation time, we reduce similarity matching on structure to simple equality matching of tokens while capturing relevant structural variations. We experimentally compare the effectiveness of our method with a wide variety of previous approaches. Besides token-based and edit-distance similarity functions, we also consider techniques based on very different concepts such as time-series analysis and information theory. To the best of our knowledge, this is the first comprehensive evaluation of effectiveness and comparison of structural similarity functions in the EM context. [1]

The rest of this chapter is organized as follows. In Section 3.1, we review several, previously proposed similarity functions for XML structures, namely: variants of TED employing a set of operations tailored to the XML data model [NJ02] and non-uniform cost model [ABG10]; an approach based on Fourier transform theory [FMM+05]; measures of structural similarity based on entropy [Hel07]; token-based methods based on the concept of *pq*-grams for ordered [ABG10] and unordered trees [ABDG08]; and simpler tokenization approaches, such as the path-shingles approach [But04]. Our own structural similarity function is introduced in Section 3.2. We present our thorough experimental evaluation in Section 3.3. The summary of this chapter is provided in Section 3.4.

3.1 Existing Approaches for XML Structural Similarity

In this section, we review previous work for measuring structural similarity of XML trees. We concentrate on the main concepts and ideas behind each approach. For more details, such as specific algorithms, we refer the reader to the respective papers.

3.1.1 Tree Edit Distance

Nierman and Jagadish [NJ02] observed that edit distance measures that allow modifications to only one node at a time can find a large distance between a pair of documents derived from the same DTD due the possibility of optional and repeated elements. To avoid this behavior, the authors devised a dynamic programming algorithm for ordered trees supporting subtree-level operations, i.e., besides the standard operations—node relabeling, node insertion, and node deletion,[2] the algorithm supports subtree insertion and subtree deletion operations. In the paper, the authors

[1]The work in [LCW07] proposes an approach to XML similarity based on Bayesian network, which bears resemblance to the collective matching strategy discussed in Chapter 2. However, this approach requires that all XML trees comply to the same schema, and, therefore, we refrain from including it in our evaluation as we focus on heterogeneous XML databases.

[2]For node insertion, only nodes without children can be inserted; for node deletion; only leaf nodes can be deleted.

consider the unit cost model. We denote by *TED-NJ* this TED variant of Nierman and Jagadish.

Furthermore, concerning semantic as well as efficiency aspects, the authors restricted the sequence of edit operations considered by the algorithm. Specifically, given a source tree T_1, a destination tree T_2, and a subtree S, the following conditions must be satisfied:

- S can be inserted into T_2 iff S already occurs in T_1, i.e., if at least one instance of S is contained in T_1—a subtree S is said to be contained in a tree T if all nodes of S occur in T, with the same parent-child relationship and the same sibling order; S can be deleted from T_1 iff S occurs in T_2.
- If S is inserted into T_2, then additional nodes cannot be inserted into S; if a node is deleted from S in T_1, then S cannot be deleted afterwards.

The first restriction addresses the matching of subtrees whose root node was defined as optional in the DTD. It also ensures the soundness of the algorithm: without this restriction, one could delete the source tree in a single operation and then insert the destination tree in a second operation. The second restriction is exploited to efficiently compute the costs of subtree-level operations in a bottom-up procedure.

The TED variant of Nierman and Jagadish [NJ02] employs a different set of operations using the unit cost model. In [ABG10], the authors described the *fanout weighted TED*, which, while using the same set of operations of the classical TED definition, employs a cost model based on node fanout. Specifically, given a constant $c > 0$, the cost model of fanout-weighted TED, *TED-F*, is defined as follows:

(a) Deletion of node u: $u_f + c$.

(b) Insertion of node v: $v_f + c$.

(c) Relabel of node u with the label of v: $\frac{u_f + v_f}{2} + c$.

Finally, we denote by *TED-U* to the classical TED formulation, e.g, the formulation using the basic set operations and unit cost model [Tai79].

3.1.2 Discrete Fourier Transformation

The structure of an XML tree can be represented as a discrete-time signal. To this end, structural nodes are encoded as real values and, then, one can interpret each visit of a node in a pre-order traversal as producing an impulse signal whose intensity is given by the corresponding node encoding. Assuming that each node is visited at a fixed time interval, we can thus obtain a time-series sequence.

Flesca et al. [FMM$^+$05] used the above approach to describe the structural information of ordered XML trees and calculate their pairwise similarity. XML documents are converted to time series by two functions: the *label encoding function* and the *tree* *encoding function*. The first function associates labels occurring in an XML collection to real values. The second function uses the label encoding to convert the sequence of labels appearing in a pre-order traversal of an XML tree to a time series. The presence of different node labels and structural patterns between XML trees is captured by the encoding functions yielding different signal shapes. Furthermore, contextual information (provided by nodes in the vicinity, e.g., parent and siblings) can be added to enrich the signal representation of nodes and substructures.

Instead of assessing the similarity of XML direct in the time domain (e.g., using time warping [YJF98] to calculate the distance between signal shapes), Flesca et al. proposed using Discrete Fourier Transform (DFT) to map the time sequences to the frequency domain. The authors argued that the comparison of structures through the analysis of their underlying frequency spectra is less sensitive to structural deviations originated by different numbers of occurrences of shared elements or by small shifts in element positions. Below, we describe the encoding functions and the DFT-based similarity function in more detail.

The label encoding function is defined by $\gamma : \mathit{labels}(C) \rightarrow \mathbb{R} - \{0\}$, where $\mathit{labels}(C)$ is the set of labels appearing in an XML collection C. Besides, function γ can also be used to encode the event of leaving a node in a pre-order traversal—we "leave" a node in a pre-order traversal after having visited its children. This can be done by spanning two symbols, l and l', for each label in collection C. The function γ is said symmetric iff $\gamma(l) = -\gamma(l')$ for each $l \in \mathit{labels}(C)$ (see [FMM+05] for alternative encodings).

More sophisticated encodings are obtained by accounting for contextual information. For example, node labels can be prepended with the label of the parent node or the complete sequence of ancestors (i.e., the full path from the root to the node). Note that the label encoding process can be viewed as assigning real values to elements of a set of structural tokens. In this sense, the tokenization method employed determines the amount of contextual information that is encoded.

Given the set of labels (or structural tokens) encodings, a pre-order traversal in a tree T defines a sequence of real values of the form $\langle \gamma(l_1), \ldots, \gamma(l_n) \rangle$. A tree encoding function enc uses this sequence to derive the time series representation of T, which is denoted by $enc(T) = \langle S_1, \ldots, S_n \rangle$. Flesca et al. proposed several tree encoding strategies; the so-called multi-level encoding was found to provide the best results. In this encoding, each element in $enc(T)$ is given by:

$$S_i = \gamma(l_i) \times B^{maxheight(C) - level(l_i)} + \sum_{l_j \in anc(l_i)} \gamma(l_j) \times B^{maxheight(C) - level(l_j)},$$

where B is the number of distinct tokens encoded by γ, $maxheight(C)$ is the maximum tree height in the collection C, $level(l_i)$ and $anc(l_i)$ are the level and the sequence of ancestors of the node associated to token l_i.

Different tree sizes induce variable-length time series, whose frequency coefficients may be incomparable. To avoid this problem, the DFT of two trees, say T_1 and T_2, is computed at M fixed frequencies, where $M = |enc(T_1)| + |enc(T_2)| - 1$, and missing coefficients are interpolated from the available ones.

The DFT-based distance, denoted by $DFTD$, is defined on basis of the approximate difference of magnitudes of the interpolated DFT ($\widetilde{DFT}$) of two encoded trees:

$$DFTD(T_1, T_2) = \left(\sum_{k=1}^{\frac{M}{2}} \left(|[\widetilde{DFT}(enc(T_1))](k)| - |[\widetilde{DFT}(enc(T_2))](k)| \right)^2 \right)^{\frac{1}{2}} .$$

3.1.3 Entropy-based Similarity

In [Hel07], Helmer adapted the *normalized compression distance* (NCD) introduced in [CV05] to XML data. This measure is attractive because it has empirically been shown to provide satisfactory effectiveness while having a runtime complexity of $O(n)$, where n is the number of symbols in a given sequence. Moreover, NCD can be calculated by using simple off-the-shelf compression tools such as GNU zip (gzip) [3]. The author also devised an alternative similarity function based on cross-parsing. In the following, we review some background on concepts behind entropy-based similarity before providing more details about the respective similarity functions.

In Information Theory, the *Kolmogorov complexity* [CT06] of an object x, denoted by $K(x)$, is the length of the shortest binary program that outputs x on a universal computer (such as the Turing universal machine). The Kolmogorov complexity of an object x can be viewed as a precise quantification of the amount of information of x and, in turn, as length of the ultimate compressed version of x. Further, the *conditional Kolmogorov complexity*, denoted by $K(x|y)$, is defined as the length of the shortest binary program that, given input y, outputs x. Based on these concepts, Bennet et al. [BGL+98] defined the notion of information distance between two objects. The normalized version of this distance, called the *normalized information distance* (NID), is given by:

$$NID(x, y) = \frac{max(K(x \mid y), K(y \mid x))}{max(K(x), K(y))} .$$

The Kolmogorov complexity provides an intuitive and principled notion of similarity, but it is not computable. Fortunately, standard compression techniques can be employed to approximate the elements in the NID formula. Cilibrasi and Vitnyi [CV05] introduced the compression-based approximation of NID, called *normalized compression distance* (NCD):

$$NCD(x, y) = \frac{C(xy) - min(C(x), C(y))}{max(C(x), C(y))} ,$$

where $C(x)$ and $C(y)$ are the compressed size of x and y, respectively, and $C(xy)$ is the compressed size of their concatenation.

[3]http://www.gzip.org/

Helmer used the gzip tool to compress the structure of XML trees. The algorithm used by gzip is a variant of the well-known LZ77 algorithm (Lempel-Ziv 1977, see Ziv and Lempel [ZL77] and [WMB99] for more details). Instead of directly compressing an XML tree, Helmer used tokenization methods to extract its structural information; the concatenation of the tokens form the sequence to which the compression method is applied. The author experimentally demonstrated that use of tokenization methods improve effectiveness, indeed. Interestingly, the best-performing tokenization method only collects the node labels of a tree (including tokens representing end tags) without actually extracting the underlying structure.

An alternative entropy-based similarity measure based on cross-parsing of two sequences was presented. Assuming that two XML trees are represented by the strings x and y, cross-parsing x with respect to y according to the algorithm of Ziv and Merhav [ZM93] proceeds as follows . First, starting from $i = 1$, find the longest prefix of x that appears as a (sub)string of y, i.e., find the largest integer m such that $x_i, x_{i+1}, \ldots x_{i+m-1} = y_j, y_{j+1} \ldots y_{j+m-1}$ for some j. If $m = 0$ (i.e., x_1 does not appear in y), we set $i = i + 1$; otherwise, $i = i + m$. The algorithm proceeds looking for the longest prefix of x, starting from position i, that appears in y until x has been parsed completely. Let $c(x|y)$ denote the number of substrings of x found in y. For example, consider the strings $x = 00110101001$ and $y = 10001010110$. Cross-parsing x with respect to y, we find the substrings 001, 10101, and 001, therefore $c(x|y) = 3$. Based on this concept, Helmer defined the following similarity measure based on Ziv and Merhav cross-parsing (*ZMC*):

$$ZMC(x, y) = \frac{c(x|y) - 1 + c(y|x) - 1}{2} \, .$$

Helmer reported the ZMC measure outperforming NCD. According to the interpretation of the author, this result is influenced by the granularity used in these measures. The gzip algorithm considers label characters as the smallest units of information in a sequence, while ZMC operates over labels. This apparent problem of using gzip can be easily solved by encoding node labes as numerical values, similar to the approach of Flesca et al. described in the previous section. We shall nevertheless consider both measures in our comparative study in Section 3.3.

Finally, Helmer proposed a hybrid approach by multiplying the values returned by the DFT function described in the previous section and ZMC. We denote this hybrid approach by *DFT-ZMC*.

3.1.4 pq-grams

The concept of *pq*-grams was introduced by Augsten et al. [ABG10] as a method to map ordered labeled trees to sets of tokens and thereafter approximate the tree edit distance via set-overlap-based measures. Informally, all subtrees of a specific shape are called *pq*-grams of the corresponding tree. This shape is defined by two positive integer values *p* and *q*. A *pq*-gram consists of an *anchor node* prepended $p-1$ ancestors, called the *stem* of the *pq*-gram, and *q* children, called the *base* of the *pq*-gram. Figure

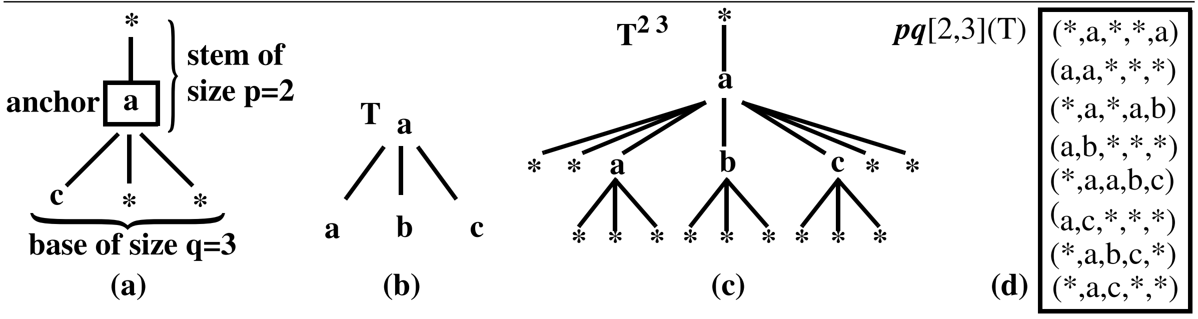


Figure 3.1: Steps for the generation of *pq*-gram tokens

3.1(a) illustrates a sample *pq*-gram for $p = 2$ and $q = 3$. The stem captures hierarchical information while the base captures sibling ordering (if $q > 1$). The concatenation of the node labels of a *pq*-gram forms a *pq*-gram token. To be able to obtain a set of *pq*-grams from any tree shape, an expanded tree $T^{p,q}$ is (conceptually) constructed from the original T by extending it with *dummy nodes*—a dummy node is a special node whose label $' * '$ is not present in any original tree. Specifically, dummy nodes are inserted as follows: $p - 1$ ancestors of the root node; $q - 1$ children before the first and after the last child of each non-leaf node and q children of each leaf node. Figure 3.1(b) and (c) show tree T and its expanded form $T^{2,3}$.

We denote by *pq*[*p*,*q*] the tokenization function that generates a set of *pq*-grams from a tree T, i.e, the $pq[p, q](T)$ profile, by collecting all *pq*-grams of the respective expanded tree. This can easily be done by performing a pre-order traversal in T. Figure 3.1(d) shows the $pq[2, 3](T)$ profile derived from $T^{2,3}$. It can be shown that the cardinality of a *pq* profile is linear in the size of the tree. Precisely, given a tree T, we have:

$$|pq[p, q](T)| = 2 \times |leaves(T)| + |nonleaves(T)| \times q - 1 \ ,$$

where $leaves(T)$ and $nonleaves(T)$ are the sets of leaf nodes and non-leaf nodes, respectively.

We denote by *PQ* the class of similarity functions defined by $\langle pq[p, q], \_, \_\rangle$.

pq-grams exhibit useful properties that agree with common intuition about hierarchical data similarity. As for any token-based method, the *PQ* similarity between two trees hinges on the number of *pq*-grams shared by their profiles. The number of *pq*-grams, in which a non-leaf node appears, depends on the number of descendants of this node within distance p. Note that higher values for p increase the number of *pq*-grams "destroyed" by edit operations such as node relabeling (recall the related discussion on *q*-grams in Chapter 2, Section 2.3.2) thereby augmenting the sensitivity to mismatches on non-leaf nodes. Besides the parameter p, the number of *pq*-grams, in which a node appears, also increases with the fanout of the node. In fact, it was shown that *pq*-grams can be used to provide a lower-bound approximation for TED using the fanout weighted cost model, i.e., TED-F. For leaf nodes, the number of corresponding *pq*-grams depends only on the parameter q. Because non-leaf nodes appear in more *pq*-grams than leaf-nodes, mismatches on them decrease the similarity to a larger extent. Finally, edit operations on nodes or structural patterns like deletion of a subtree only affect *pq*-grams within a relatively small neighborhood. Hence, modifications concentrated on a specific part of a tree have less impact than those dispersed over the whole tree.

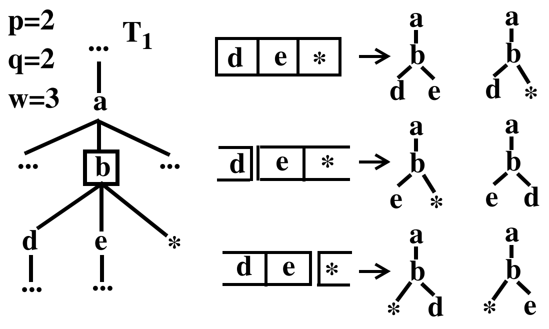
3.1.5 Windowed pq-grams

So far, we have discussed similarity functions for ordered trees. The similarity measure we consider in this section—as well as the similarity functions discussed in the following sections—is designed for unordered trees. In general, edit distances are inappropriate for unordered trees. When sibling ordering is disregarded, finding the minimum-cost edit sequence is computationally intractable, because all sibling permutations must be considered. Token-based similarity functions are popularly used to calculate the similarity between unordered.

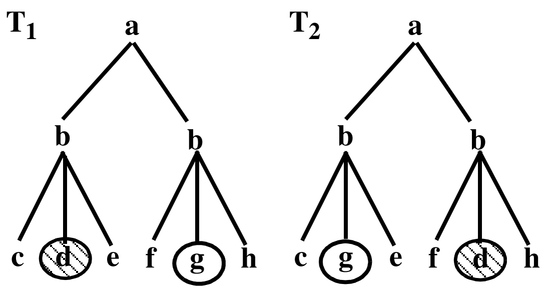
Augsten et al. [ABDG08] presented a variant of the *pq*-gram concept for unordered trees, the so-called *windowed pq-grams* (*wpq*-grams, for short). In the original *pq*-gram definition, sibling ordering is captured by the base component. A straightforward approach would generate *pq*-grams for all possible bases, i.e., given an anchor node u, generate $\binom{u_f}{q}$ *pq*-grams. This approach is unsatisfactory, because the number of *pq*-grams in which a node appears increases too sharply with its fanout and, thus, nodes with larger fanout are overemphasized. The approach based on *wpq*-grams alleviates this shortcoming while satisfying important properties as discussed below.

wpq-grams have the shape of *pq*-grams: they consist of a stem and a base, as illustrated in Figure 3.1(a), whose sizes are determined by the parameters $p > 0$ and $q > 0$, respectively. The main difference is the token generation process, which requires the parameter $w > q$ determining the window length.

The tokenization function generating *wpq*-grams (denoted by *wpq*[w,p,q]) performs tree steps: a) sorting the unordered tree; b) extending the sorted tree with dummy nodes; and c) computing the *wpq* profile on the extended tree. Step a) consists of imposing a total order on the nodes, e.g., the lexicographical order according to the node labels, and using this order to sort the children of all non-leaf nodes (ties have no effect on the final *wpq* profile and, therefore, can be broken arbitrarily). In step b), dummy nodes are inserted as ancestors of the root node and as children of the leaves in the same way as for *pq*-grams. For each non-leaf node u, such that $u_f < w$, $w - u_f$ dummy nodes are inserted after the last child. *wpq*-grams are obtained in step c) by sliding a window of length w over the children of non-leaf nodes. The window is wrapped around the right border, i.e., the last children are followed by the first children in the last $w - 1$ windows. At each window, a set of bases is produced by taking the first window node and all combinations of the remaining $w - 1$ nodes, totaling $\binom{w-1}{q-1}$ bases per window position. The node order in the bases is preserved



(a) *wpq*-gram generation



(b) Detection of node moves to other parents

Figure 3.2: Example of *wpq* profile generation and varying sibling sets between parents with the same label

and each base is prepended by the stem to form a *wpq*-gram. Figure 3.2(a) illustrates this process in tree T_1 (Figure 3.2(b)) for the anchor node with label b. For leaf-nodes, a single base is formed by q dummy nodes.

Similarity functions using *wpq* as tokenization function are denoted by *WPQ*. The cardinality of the *wpq* profile generated from an unordered tree T is given by:

$$|wpq[w, p, q](T)| = (|T| - 1) \times \binom{w-1}{q-1} + |leaves(T)| \,.$$

A distinctive feature of the *wpq*-gram approach is the exploitation of sibling relationships for enhancing the sensitivity to structural deviations. In particular, WPQ is able to capture varying sibling sets between parents with the same label. For example, consider the (sorted) trees T_1 and T_2 in Figure 3.2(b). The parents of d and g are swapped in the two trees. For $q > 1$, T_1 and T_2 deliver different profiles and, therefore, the similarity is decreased. In general, such structural deviation is ignored by the path-based similarity functions that we will cover shortly.[4] Another aspect of *wpq*-grams is that the set-overlap-based similarity (e.g., the Jaccard similarity) between a set of bases generated for two nodes approximates the similarity of their underlying sibling set. Augsten et al. showed that bases of size 2, i.e., $q = 2$, provide the closest approximation while being able to capture node moves to other parents.

3.1.6 Further Tokenization Methods

We further consider two simple path-based tokenization functions for unordered trees. The first tokenization function, denoted by *pshingle*, produces a set of the so-called *path shingles* [But04], an adaptation of *shingles* [Bro97] for tree-structured data.

[4]Note that, for $q > 1$, *pq*-gram similarity can be viewed as a path-based similarity function.

In a nutshell, the path-shingles approach generates tokens by prepending each element node in a tree with the full path from the root to this node.

Example 3.1. *Consider subtrees a) and b) shown in Figure 2.4. Their respective profiles of path shingles are:*

$pshingle(a)$ $=\{$`exame`, `exame` $\circ$ `patient`, `exame` $\circ$ `patient` $\circ$ `study`, `exame` $\circ$ `patient` $\circ$ `study` $\circ$ `id`, `exame` $\circ$ `patient` $\circ$ `study` $\circ$ `description`, `exame` $\circ$ `patient` $\circ$ `name`, `exame` $\circ$ `patient` $\circ$ `relatives`, `exame` $\circ$ `patient` $\circ$ `relatives` $\circ$ `mother`$\}$,

$pshingle(b)$ $=\{$`exame`, `exame` $\circ$ `study`, `exame` $\circ$ `study` $\circ$ `patient`, `exame` $\circ$ `study` $\circ$ `patient` $\circ$ `name`, `exame` $\circ$ `study` $\circ$ `patient` $\circ$ `mother`, `exame` $\circ$ `study` $\circ$ `id`, `exame` $\circ$ `study` $\circ$ `description`$\}$.

The other tokenization function we regard consists of simply collecting all *root-to-leaf* paths of a tree T; this function is denoted by *rtl*. Similarity functions defined by $\langle pshinle, \_, \_\rangle$ and $\langle rlt, \_, \_\rangle$ are denoted by *PSH* and *RTL*, respectively. The main motivation for including these simple similarity functions in our approach is to contrast them with our path similarity approach, which is described in the next section.

3.2 Structural Similarity Based on Path Clustering

We now present our approach for measuring the structural similarity between XML trees. Our measure pertains to the class of token-based similarity functions. As such, its key ingredient is the method for extracting structural information and represent it as a set of tokens. We consider unordered trees and, therefore, the direct competitors of our similarity function are *wpq*-grams and the path-based tokenization methods, which were presented in the last two sections.

The main novelty of our technique is the exploitation of path synopses and clustering methods to derive compact structural surrogates from XML data. Our approach addresses two drawbacks with respect to effectiveness and efficiency that we identified in previous work:

- *Lack of support for approximate path matching*: The ability of *approximately* matching navigational paths is of paramount importance for dealing with structural heterogeneity. Most structural deviations result in different hierarchical nestings that encode and lead to the same information. This happens very frequently in heterogeneous datasets where documents often exhibit divergent data arrangements. Moreover, most common structural changes in evolving scenarios consist of increasing the structure [Sed05]. For example, a simple element, say `author_name`, can evolve to a complex element containing `first_name` and `last_name` as sub-elements. Such structural changes are in accordance with the "schema-later" practice [JCE+07], which is one of the main motivations for using XML: data is first loaded into the database, and structure is incrementally added as more insight about the data is obtained. In such cases, we may have the situation of different paths relating to the same piece of information.

In some sense, we can interpret such paths as fuzzy duplicates. Consider the subtrees illustrated in Figure 2.4. The paths $/exam/patient/relatives/mother$ in subtree $a)$ and $/exam/study/patient/mother$ in subtree $b)$ encode the same information and can, therefore, be viewed as duplicates. In fact, the whole trees are duplicates and there is a one-to-one correspondence between their path sets. Thus, an intuitive approach for measuring the similarity of trees is to identify pairs of duplicate paths between them. Note that none of the previously described token-based similarity functions would yield a high similarity value for subtrees $a)$ and $b)$. Although hierarchical information is captured by these approaches (e.g., the stem component in *wpq*-grams and path shingles), their underlying tokenization methods are still too ″coarse“ for accurate path matching. The main reason is that tokens in these methods encode partial paths, i.e., sequences of two or more nodes; however, node-level granularity is necessary for matching the paths in Figure 2.4. The problem with tokens based on partial paths is analogous to that of using a large q-gram size that we alluded to in Chapter 2: a disproportionate number of tokens are destroyed by node mismatches. Here, this problem is exacerbated because paths in subtrees are usually shorter than strings.

- *Repeated structural comparisons*: Most XML data is characterized by the presence of highly repeated substructures. Indeed, it is well known that, for many real-world data, compression techniques are able to reduce the size of the structural part of XML documents to less than 10% of their original size [BGK03]. Moreover, such structures are likely to appear many times across different XML documents, even for heterogeneous datasets. This fact is ignored by all similarity approaches discussed so far. Hence, employing these methods in a similarity join predicate implies that the similarity between repeated structures is computed anew every time these structures show up during join processing.

We address the first shortcoming by adopting a structural similarity measure based on approximate path matching. The foundational path similarity function employs a node-level tokenization method, thereby favoring accurate path similarity assessment. Paths are different in nature from strings because the hierarchical nesting imposes an order on the path elements. We capture this ordering in a weighting scheme that assigns weights to path elements according to their nesting level in a monotonically decreasing way. As a result, we are able to obtain meaningful results for frequent types of path variations. In addition to providing an effective similarity measure for XML structures, we will show in Chapter 4 that our path-similarity-based approach facilitates the combination of textual and structural information as well as the design of tailor-made structures for entity description selection.

We avoid repeated computation of path similarity by following a ″one-for-all“ approach. We first compute the similarity between all paths in a path synopsis. Concisely, a path synopsis is a structural summary of all paths appearing in an XML collection (we provide a detailed description of a path synopsis later in this chapter).

This step can be performed very efficiently. First, the number of paths in a path synopsis is in most cases orders of magnitude smaller than in the data instances. Second, because we employ a token-based method for path similarity calculation, we can use a set similarity join algorithm comparable to the one employed to join whole XML trees to obtain pairwise similarity values (see Chapter 5). The output of the similarity join is used by a clustering method to group similar path classes. Finally, the resulting cluster information is used to generate a token-based representation of documents. As a result, when joining XML trees, similarity matching between paths is reduced to simple equality matching of tokens, i.e., tokens with the same value represent duplicate paths, that have been previously identified in the pre-processing step. Moreover, our approach produces very compact structural representations, which are bounded by the number of paths.

In rest of this section, we first describe the underlying path similarity function. Then, we give details about the path synopsis structure, before we describe our cluster-based approach to generate structural representations. We consider different weighting strategies for structural tokens and, finally, we discuss some related approaches.

3.2.1 Path Similarity Function

In this section, we propose a token-based path similarity function. To this end, we define a simple tokenization function called *path* that splits a tree path into a set of tokens by converting each element node label into an annotated token. The motivation to use annotated tokens is different from that discussed in Section 2.3.2. Here, token annotation serves to deal with paths containing element recursion, e.g., paths containing multiple occurrences of the same node label. When calculating the overlap between two *path* profiles, the matching of annotated tokens derived from recursive labels is done from low to high nesting levels.

Example 3.2. *Consider two paths* p_1 =/a/a/b *and* p_2 =/a/b/b*. Their path profiles are* $path(p_1)$ $=\{\mathtt{a} \circ 1,\ \mathtt{a} \circ 2,\ \mathtt{b} \circ 1\}$ *and* $path(p_2)$ $=\{\mathtt{a} \circ 1,\ \mathtt{b} \circ 1,\ \mathtt{b} \circ 2\}$*. Therefore, we have* $path(p_1) \cap path(p_2) = \{\mathtt{a} \circ 1, \mathtt{b} \circ 1\}$.

For ease of notation, we omit from now on the token annotation from *path* profiles, i.e., we refer to the annotated profile $path(p) = \{\mathtt{a} \circ 1, \mathtt{a} \circ 2\}$ simply as $\{\mathtt{a},\ \mathtt{a}\}$.

Further, a weighting scheme is applied to express the relative node significance in a path. In hierarchically structured data, more general concepts in the corresponding domain are normally placed at lower nesting depths. Mismatches between two paths on such low-level concepts may suggest that the information contained in them ls semantically more “distant”. Therefore, an intuitive heuristics is to assign higher importance to nodes at lower nesting depths. Finally, given two paths represented as weighted sets, their similarity can be assessed using a set-overlap-based similarity function. Next, we formally define these concepts.

Definition 3.1 (Level-based Weighting Scheme (LWS))**.** *Let* p *be a path and* $path(p) = \{\mathtt{t_0}, \ldots, \mathtt{t_n}\}$ *be its profile, where token* $\mathtt{t_i}$ *relates to an element node at nesting level* i*. The* *level-based weighting scheme assigns a weight to each token in* $path(p)$ *producing the weighted profile* $LWS(path(p)) = \{\langle \mathtt{t}_0, lws(t_0)\rangle, \ldots, \langle \mathtt{t}_n, lws(t_n)\rangle\}$ *where:*

$$lws(t_i) = e^{\beta i} , \tag{3.1}$$

and $\beta \leq 0$ *is a decay rate constant.*

We refer to the class token-based similarity functions employing the LWS-weighted *path* profiles, i.e, defined by $\langle path, LWS, \_\rangle$, as the *Weighted Path Similarity* (WPS).

Example 3.3. *Consider* p_1 =/`patient`/`relatives`/`mother` *and* p_2 =/`study`/`patient`/`mother` *be two paths appearing in the document shown in Figure 2.4. Applying LWS with decay rated* $\beta = -0.1$ *to the path profile of* p_1 *and* p_2*, we obtain the following weighted profiles:*

$LWS(path(p_1))$ =$\{\langle \mathtt{patient}, 1\rangle, \langle \mathtt{relatives}, 0.904\rangle, \langle \mathtt{mother}, 0.818\rangle\}$,

$LWS(path(p_2))$ =$\{\langle \mathtt{study}, 1\rangle, \langle \mathtt{patient}, 0.904\rangle, \langle \mathtt{mother}, 0.818\rangle\}$.

The two profiles share the tokens `patient` *and* `mother`*. Mother has weight* 0.818 *in both profiles, whereas, for the token* `patient`*, the weight value of* 0.904 *is returned by the minimum function. Thus, the weighted overlap is* 1.723*. The weight of the profiles* $LWS(path(p_1))$ *and* $LWS(path(p_1))$ *are both* 2.723*. Thus, using WJS as set-overlap similarity measure, we have* $WPS(p_1, p_2) = 1.723/(2.723 + 2.723 - 1.723) = 0.462$.

The use of the minimum function ensures that when matching two tokens t_i and t_j, which are related to two nodes at nesting level i and j, where $i < j$, the lower weight value of t_j is used in the calculation of the weighted overlap. Also, differently from similarity functions based on *q*-grams for $q = 1$, LSW together with the minimum function avoids having maximum similarity for two paths that correspond to different permutations of the same set of node labels.

Finally, we observe that, owing to the strictly decreasing weighting rule of *LWS*, *WPS* may yield non-intuitive results, when applied to long paths. For example, depending on the value of decay rate β, two long paths sharing a smaller portion of element nodes at higher nesting levels, but having a larger part of unrelated nodes at lower levels, will still have relatively high similarity according to *WPS*. One solution could make the weights constant from some level on. Then, the weighted norms in the denominator of a set-overlap-based similarity function like Jaccard would be kept sufficiently large to decrease the final result. On the other hand, empirical studies provide evidence that the vast majority of XML data worldwide has less than 8 levels [HMS07] and, therefore, the above effect is hardly an issue in practice.

3.2.2 The Path Synopsis

As pointed out in the beginning of this chapter, when XML is used to represent semi-structured data, the inherent complexity of the underlying structure prevents the specification of a fixed schema in advance. Nevertheless, some knowledge about the data structure is badly needed to enable users to formulate meaningful queries over the data and the query optimizer to identify efficient query execution strategies.

Path synopsis (PS) is a tree-structured index for providing and maintaining a structural summary of XML data [Mat09]. It can be viewed as a dynamic schema generated from a given XML collection. All distinct paths appearing in a collection are described exactly once in the PS, which is incrementally maintained as the XML collection is modified. PS has also been referred to in the literature as DataGuide [GW97] and *1*-Index [MS99]. The following definition of PS is based on that of Mathis [Mat09], slightly modified to adhere to the data model presented in Chapter 2.2.1 and to the purposes of our work.

Definition 3.2 (Path Synopsis (PS)). *Let C be a collection of XML trees. The path synopsis PS of C is a tree structure that satisfies the following conditions:*

1. *Let u_n be a node in PS and $\langle u_0, u_1, ..., u_n \rangle$ be the path from the root node of the PS to u_n. Then, there is a least one tree $T \in C$ containing an element node v_n such that the path from the root node of T to v_n is $\langle v_0, v_1, ..., v_n \rangle$ and $u_i = v_i$ for $0 \leq i \leq n$.*

2. *Let v_n be a node in a tree $T \in C$ and $\langle v_0, v_1, ..., v_n \rangle$ be the path from the root node of T to v_n. Then, there is exactly one node u_v in PS such that the path from the root node of PS to u_n is $\langle u_0, u_1, ..., u_n \rangle$ and $u_i = v_i$ for $0 \leq i \leq n$.*

The first condition ensures that a PS encodes no path that does not appear in C, while the second condition ensures that every path appearing in C is encoded in the PS exactly once. For example, the corresponding path synopsis of the document shown in Figure 2.4 (starting from the `exam` node) is depicted in Figure 3.4 (disregard the node identifiers and the accompanying table for the moment). Note that the PS is an unordered tree structure: information about sibling ordering is neither maintained nor relevant. We refer to the paths in a PS as *path classes*; paths appearing in trees of the XML collections are referred to as *path instances*. Given a path instance p, we denote by $class(p)$ its corresponding path class.

In the absence of the PS structure, the paths can be collected in a single pass over the data—this scan step can be coupled with the process described in the next section. In XML-enabled relational systems, a popular approach is the so-called *shredding*, i.e, the mapping of XML trees to several relational tables. Paths are frequently used as the unit of decomposition in these approaches [YASU01] and stored into *path tables*, which can be readily used to derive the PS structure. In distributed environments, PSs can be exported by local sources to a central component, e.g., the mediator module of an EII system, and merged to form a global PS. Finally, in the dataspace paradigm, the global PS can constructed in a "pay-as-you-go" fashion. Further details about the construction and maintenance of PS structures is beyond the scope of this thesis. Henceforth, we assume that a PS structure is available for a given XML collection. PS structures are instrumental for efficient XML query evaluation and are, therefore, pervasively supported by XML DBMSs. Indeed, PS is a fundamental data structure in XTC [Mat09], which is the target platform of our similarity join framework.

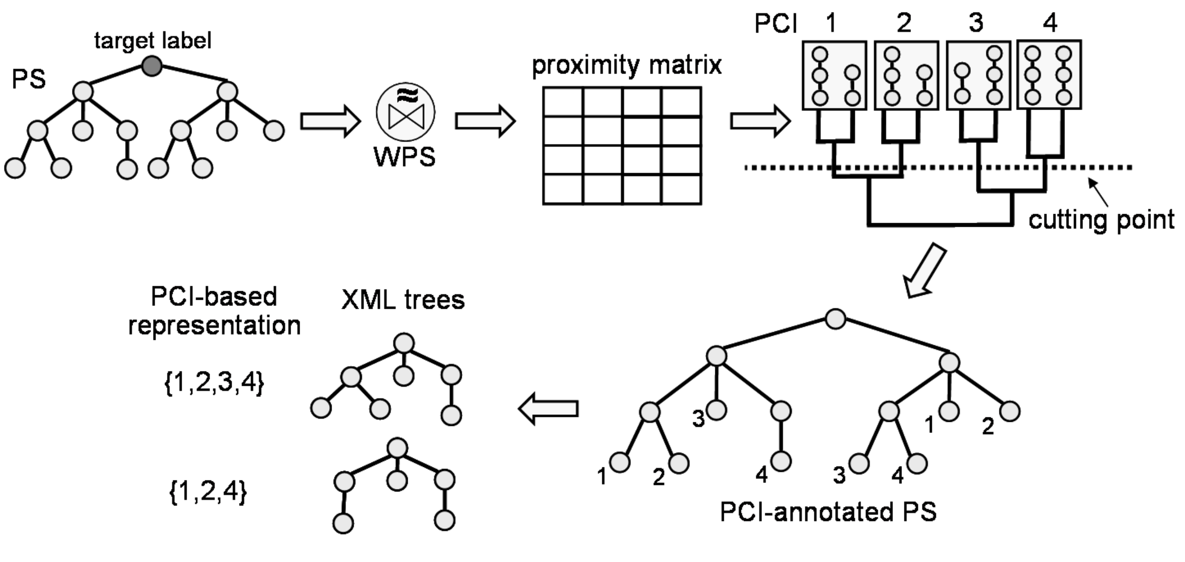


Figure 3.3: PCI generation process

3.2.3 XML Representation Based on Path Clustering

We now present our approach to generating compact surrogates for the structure of XML trees. In a pre-processing step, the WPS similarity function is employed in a cluster algorithm to group "duplicate paths" in a PS. In a sense, we can view this step as an EM subprocess where the entities to be matched are path classes. We then (conceptually) merge the path duplicates by assigning the same identifier to path classes belonging to the same group. We call these identifiers *Path Cluster Identifiers* (PCIs). The paths in a PS are then annotated with the corresponding PCIs (again, conceptually). Finally, the PCI-equipped PS is used to support the generation structural token sets.

Figure 3.3 depicts the process of generating PCI-based representations of XML trees. Given a PS of an XML collection, we start by specifying a target label *tgl*, corresponding to the entity to be matched (e.g., `exam` in Figure 2.4). Let P^{tgl} be the set of all path classes relative to *tgl*, i.e., (partial) paths in the structural summary having *tgl* as root label. In case of nested occurrences of *tgl*, we consider only the paths rooted by the topmost occurrence. We then apply a self-similarity join on P^{tgl} with predicate $WPS(p_1, p_2) \geq \tau$. We use the output of the similarity join to construct a *proximity matrix* containing all pairwise similarity values of the path classes in P^{tgl} (for pairs not satisfying the similarity join predicate, we assign a similarity value of 0). This similarity matrix is the input to a cluster method that generates a set of path clusters (partitions). In this thesis, we use the UPGMA *Agglomerative Hierarchical Clustering* method with a user-specified threshold as cutting point in the dendrogram [JD88].[5]

[5]Note that the specific clustering method is orthogonal to all the methods used in this paper (with

We denote by PC_θ^{tgl} the set of path clusters generated from P^{tgl} at cutting threshold θ. All path clusters are numbered with integer values, i.e., PCIs. Finally, we annotate the path classes $p \in PS$ with the corresponding PCI. For ease of notation, let i be the corresponding PCI of a path cluster $pc^i \in PC_\theta^{tgl}$. Figure 3.4 shows the PS of of the document illustrated in Figure 2.4 equipped with PCIs, for tgl = `exam`, $\theta = 0.6$, and decay rate of $\beta = -0.1$ for LWS. The values in the box on the left are the similarity values at which the clusters were formed.

After having equipped the PS with PCIs, we are able to easily derive a structural tokenization function. For this purpose, we decompose a tree into a set of paths and, for each path p, the corresponding PCI can be obtained from the annotated PS and used as a structural token. As a consequence, the structure of the tree is represented as a set of *pci tokens*, where each token denotes the appearance of a path instance related to a path cluster. We denote by *pcl* this tokenization function based on path clusters, which is formally defined as follows.

Definition 3.3 (*pcl* Tokenization Function)**.** *Let $PC = \{pc^1, \ldots, pc^n\}$ be set of path clusters. Given a path instance p, we say that $p \in pc$ iff $class(p) \in pc$. Let T be a tree and $rlt(T) = \{p_1, \ldots, p_n\}$ be the its set of of root-to-leaf paths. The pcl tokenization function generates a profile from the T as follows:*

$$pcl(T) = \{i_1, \ldots, i_n : p_k \in pc^i, 1 \geq k \geq n\}$$

As usual, we generally denote by *PCL* a similarity function of the class defined by $\langle pcl, \_, \_\rangle$.

Example 3.4. *Consider subtrees a) and b) in Figure 2.4. The pcl profiles of subtrees a) and b) according to the PCI-annotated PS in Figure 3.4 are both $\{1, 2, 3, 4\}$. Thus, the similarity value of $PCL(a, b)$ according to any set-overlap-based similarity function is maximum, i.e., 1.*

In the example above, note that subtrees a) and b) have maximum similarity even though they have no path in common. This observation highlights a salient feature of our PCI-based representation: equality matching of single tokens incorporates similarity matching of whole paths for free. The actual path comparison is done only once during the clustering process thereby avoiding repeated path similarity computations when evaluating the similarity join operation.

At a high level of abstraction, we can interpret our approach as a *hash-based similarity matching* method [Ste07]. Specifically, the generation of PCIs from path sets can be viewed as a *distance-preserving hashing function*—such hash functions form the basis of the widely used LSH algorithm for probabilistic similarity search [GIM99]: similar paths are mapped to the same integer values. Drawing a parallel, the LSH employs an embed-project-hash paradigm, whereas, here, we follow an EM-based approach for hashing tree paths.

minor modifications). Other techniques such as K-means and DBSCAN are also applicable.

PCI	θ
1	0.745
2	0.745
3	0.745
4	0.633

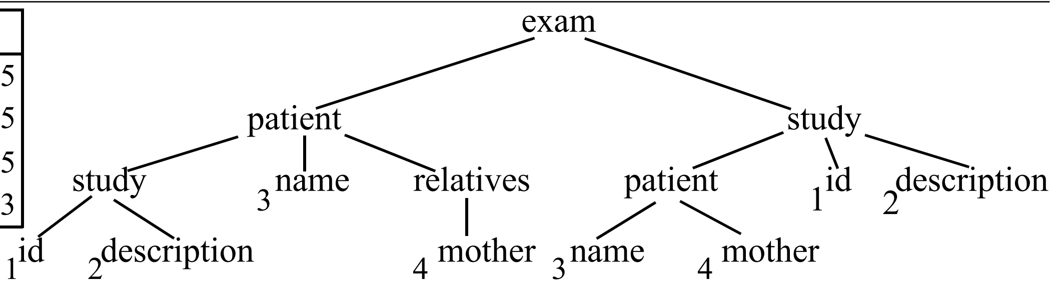


Figure 3.4: PS equipped with PCIs

Finally, because the PS is a tree structure, the matching of paths in the PS can be done efficiently by standard tree traversal algorithms. In addition, besides supporting query formulation and optimization, PS has also been exploited for designing a space-economic storage model for XTC [HMS07]. In Chapter 6, we describe how we take advantage of this storage model to obtain PCI values without even having to access the PS structure.

3.2.4 Structural Weighting Strategies

We now discuss the weighting scheme used for structural tokens. As stated in Section 2.3.2, we use annotated tokens in token-based similarity functions, which rules out the TF weighting schemes. Also, in the beginning of this chapter, we have already argued about the importance of accounting for subtree occurrence mismatches in EM tasks. Similar to strings, using the IDF weighting scheme on structural tokens emphasizes frequency disagreements, because the frequency of occurrences of annotated structural tokens in a collection decreases monotonically, i.e., $\mathit{freq}(t \circ 1, C) \geq \mathit{freq}(t \circ 2, C)$, and, therefore, their IDF weight increases monotonically. As a result, for our PCI-based tokenization method, the similarity between trees exhibiting a considerable difference in the number of occurrences of a path would be markedly reduced. We can view weighting structural tokens using IDF as an opposite approach to those proposed for the CDI problem, which aim at disregarding frequency divergences of substructures. A moderate approach would be not to employ a weighting scheme, i.e., to consider each token uniformly. Finally, we can work on multi-sets of tokens (by not annotating the tokens) and use the TF weighting scheme to dampen the effect of token frequency in trees. In Section 3.3, we empirically evaluate different weighting strategies on all structural tokenization methods described in this chapter.

3.2.5 Related Work

Vinson et al. [VHdSdM07] present an XML path similarity function based on SED. Node labels are used as units of information and the comparison of each label unfolds a second SED computation as sub-operation, whose result is then used to derive the cost of label substitution; node level information is disregard. Note that we could easily make WPS sensitive to label similarity by decomposing each path label into sets of q-grams. Although this strategy would enable us to capture syntactic variations on node labels—as the proposal of Vinson et al. [VHdSdM07]—we believe that most variations at the schema level are semantic rather than syntactic. Hence, more complex schema matching techniques are nevertheless necessary to uncover semantic correspondences.

Dalamagas et al. [DCWS06] exploited structural summaries to cluster XML documents by structure. Their approach entails extracting a structural summary from all XML documents in a collection, using a tree edit distance algorithm on the summaries to calculate all pairwise distances, and, finally, applying a clustering algorithm to group similar documents. Here, we exploit the combined structural summary of all documents in a collection, i.e., the PS, instead of the summary of each tree in isolation. Joshi et al. [JAKN03] employed the bag-of-tree-paths model, which represents tree structures by a set of paths. In this work, paths are exactly matched; in contrast, we match paths approximately. Further, our aim is completely different from these two previous approaches. We do not cluster XML documents directly; rather, we cluster paths to derive compact structural representations that can be, afterwards, combined with textual representations to calculate the overall tree similarity (see Chapter 4).

As already mentioned, exploitation of structure has been extensively investigated in XML search [KMdRS06]. A common approach is to approximately match the structural constraints expressed in the query with the context of specific nodes. In [CMM+03], fuzzy and partial match of term contexts (root-to-leaf paths) are supported. In this approach, path similarity has to be computed every time a query is evaluated, whereas, in our approach, all path similarity calculations are computed only once in the pre-processing step.

3.3 Experiments

In this section, we describe extensive empirical experiments performed for all structured similarity functions discussed in this chapter. Besides comparing the accuracy of a highly representative set of structural similarity functions, we aim at evaluating to what extent we can identify duplicates by solely relying on structural information. Finally, we also analyze the performance of several different weighting schemes for token-based structural similarity functions.

3.3.1 The Competitors

Table 3.1 shows information about the structural similarity functions considered in our experiments. We evaluate 12 measures to assess the similarity between XML trees: 8 of them employing similarity functions designed for ordered trees (TED-U, TED-F, PQ, TED-NJ, NCD, ZMC, DFTD, and DFT-ZMC) and 4 designed for

Table 3.1: Structural similarity functions

Similarity Function	Description
TED-U (T-U)	Unit cost TED for ordered trees (Section 2.3.1).
TED-F (T-F)	Fanout weighted TED for ordered trees, $c = 1$ (Section 3.1.1).
TED-NJ (NJ)	TED variant of Nierman and Jagadish for ordered trees (Section 3.1.1).
NCD	Normalized compression distance applied on ordered trees (Section 3.1.3); tokenization function: *tags*.
ZMC	Ziv and Merhav crossparsing applied on ordered trees (Section 3.1.3); tokenization function: *labels*.
DFTD (DFT)	Similarity function for ordered trees based on Fourier transform theory (Section 3.1.2); tokenization function: *parchild*, tree encoding: multi-level.
DFT-ZMC (D-Z)	Similarity function for ordered trees resulting from the multiplication of the values returned by DFT and ZMC (Section 3.1.2).
PQ	Token-based similarity function for ordered trees defined by $\langle pq[p = 2, q = 2], Jaccard, \epsilon\rangle$ (Section 3.1.4).
WPQ	Token-based similarity function for unordered trees defined by $\langle wpq[w = 3, p = 2, q = 2], Jaccard, \epsilon\rangle$ (Section 3.1.5).
PCL	Token-based similarity function for unordered trees defined by $\langle pcl, Jaccard, \epsilon\rangle$ (Section 3.2), decay rate of 0.1 for *LWS* and a threshold of 0.4 as cutting point for the dendrogram.
PSH	Token-based similarity function for unordered trees defined by $\langle pshinle, Jaccard, \epsilon\rangle$ (Section 3.1.6).
RTL	Token-based similarity function for unordered trees defined by $\langle rtl, Jaccard, \epsilon\rangle$ (Section 3.1.6).

unordered trees (PCL, WPQ, PSH, and RTL); all of them have already been discussed in the previous sections. For ordered trees, the last 5 similarity functions were originally proposed to tackle the CDI problem; the remaining similarity functions, for ordered and unordered trees, address no particular application scenario. For most functions, we used the parameter setting which provided best performance in the original papers. The exceptions are DFT- and entropy-based similarity functions. These functions were proposed and evaluated in the context of the CDI problem. In the EM context, however, we observed divergent results. For example, in [Hel07], several tokenization functions were considered to extract structural information from XML trees; *rtl* (see Section 3.1.6) was shown to outperform *labels*, which simply collects all element node labels of a tree in a bag of labels. However, we found rtl performing very poorly in our experiments. Therefore, for similarity functions involving DFT and entropy, we report the results using the best parameter setup we identified. Note that we used unweighted profiles for all token-based similarity functions; weighting schemes are compared in Section 3.3.5. Finally, some functions are abbreviated in the experimental charts to save space; the abbreviations are given between parentheses after the name of the corresponding similarity function in Table 3.1.

3.3.2 Datasets

For our empirical study, we use four real-world XML datasets: *Nasa*[6] containing astronomical data, two protein sequence databases *SwissProt*[7] and *PIR-PSD*[8] (*PSD* for short), and *DBLP*[9] containing information about computer science publications. Detailed information about the datasets is given in Table 3.2 (notice that *sdev* corresponds to *standard deviation*). For all of them, we obtained sets of XML documents by deleting the root node of each XML dataset. The resulting documents are structurally very heterogeneous. Nasa and SwissProt have much richer and irregular structure than PSD and DBLP. Nasa contains the largest (w.r.t. number of nodes) and widest (w.r.t. number of paths) trees. Nasa also exhibits very dispersed cardinality distribution, in terms of both nodes and paths per tree (i.e., large standard deviation). Additionally, the paths in Nasa are the deepest in average. On the other hand, SwissProt has the largest number of distinct node labels and about 2.5x more distinct paths than Nasa. PSD is structurally simpler than the previous two datasets; nevertheless, it has almost the same number of distinct labels and paths as Nasa and its average path depth is very close to that of SwissProt. Finally, the structure of DBLP is by far the simplest and most regular.

[6]http://www.cs.washington.edu/research/xmldatasets/
[7]http://us.expasy.org/sprot/
[8]http://pir.georgetown.edu/
[9]http://dblp.uni-trier.de/xml/

Table 3.2: Dataset statistics

Dataset	*Nasa*	*SwissProt*	*PSD*	*DBLP*
Description	Astronomical data	DB of protein sequences	DB of protein sequences	Computer science index
Target label	`dataset`	`Entry`	`ProteinEntry`	`inproceedings`
no. subtrees	2435	50000	262525	689438
nodes per tree mean/sdev/median	218.8/258/169	103.3/66.3/89	85/40.8/78	12.4/3.8/12
# distinct labels	68	97	65	21
# distinct paths	73	191	72	28
paths per tree mean/sdev/median	153.5/169.2/119	84/52.7/72	64.8/34.8/58	11.4/3.84/11
path depth mean/sdev/median	3.6/1.3/4	2.4/0.6/3	2.5/0.9/2	1.7/0.9/1

3.3.3 XML Fuzzy Duplicate Generation

From all datasets, we derived several datasets containing fuzzy duplicates by creating exact copies of the original trees, removing text nodes, and then performing structural transformations on their structure. The transformations aim at simulating typical structural deviations between fuzzy duplicates appearing in ad-hoc and heterogeneous datasets, divergences resulting from schema evolution and versioning [Sed05, MML07], and the inherent structural heterogeneity that naturally emanates from the XML data model. We support the following transformations:

- *Insert Node*: given a node u and integers i and j, where $i < j$ and $j < f_u$, inserts a node v as the *i*-th child of u; the children from i to j become children of v.
- *Delete Node*: deletes the *i*-th child of a node u; all children of the deleted node become children of u starting from position i.
- *Relabel Node*: changes the node's label for another label appearing in the dataset.

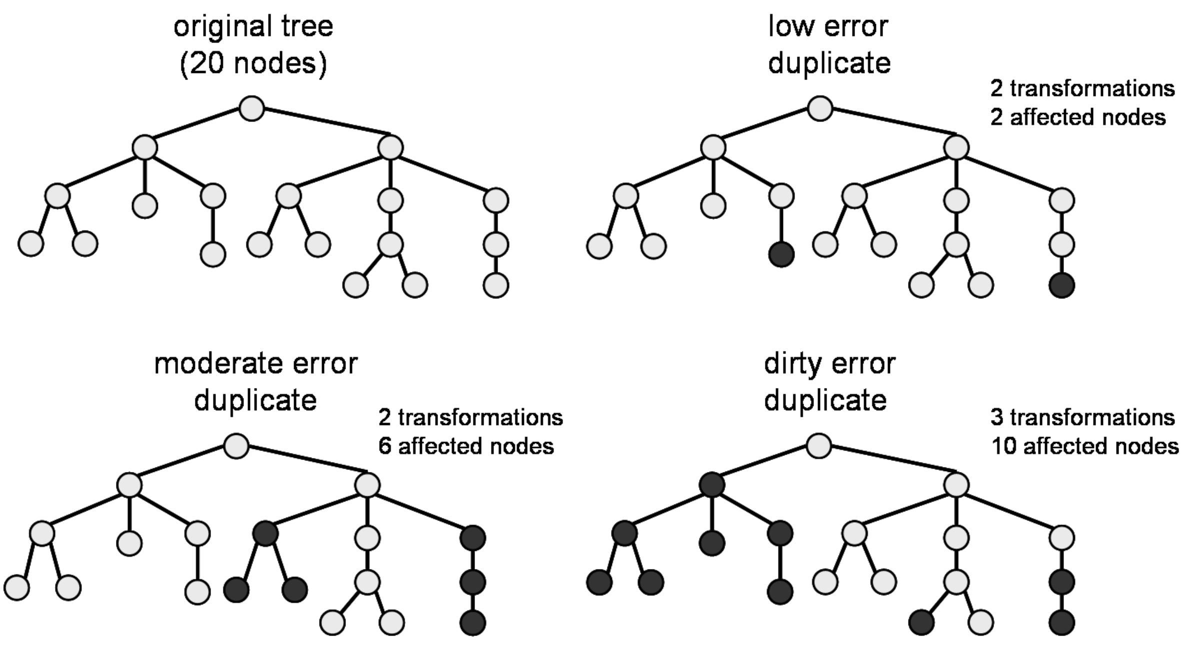


Figure 3.5: Generation of low, moderate, and dirty error duplicates

- *Invert Node*: switches the position between a node and its parent.
- *Delete Path*: deletes the path from a node down to the leaf.
- *Delete Subtree*: deletes a node and all its descendants.
- *Swap Subtrees*: swaps the position of two sibling nodes (and their corresponding subtrees).

The first three transformations follow the semantics of the classical set of TED operations [Tai79]. The Invert Node transformation aims at representing different hierarchical organizations as illustrated in Figure 2.4 on Page 37; Delete Path and Delete Subtree simulate structural heterogeneity owing to optional and repeated sub-elements. Finally, Swap Subtrees leads to different permutations in ordered trees.

To restrict the degree of modification applied on a tree, we define *error extent* as the percentage of nodes from a subtree which are affected by a set of structural transformations. We considered as affected the node that received the modification as well as all its descendants. We classify the fuzzy copies generated from each data set according to the error extent: we have low (10%), moderate (30%), and dirty (50%) error datasets. Figure 3.5 illustrates the generation of low, moderate, and dirty error duplicates. This methodology allows evaluating the robustness of the similarity functions under study. As the error extent increases, duplicate trees become more distant from the original tree and may become more close to trees related to different entities.

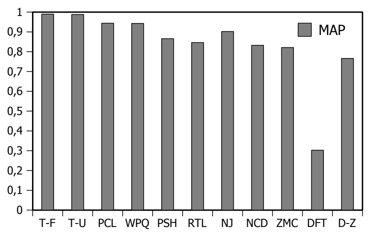
Therefore, by varying the error extent, we can study how well the similarity functions "tolerate" the deviations caused by the set of transformations while still being able to assign higher similarity to duplicates than to non-duplicates. Notice that the robustness of the similarity functions is also dependent of the original dataset, i.e., how structurally separated are the trees in the source datasets.

Transformations are randomly selected and sequentially performed until the desired error level is reached. For each transformation, we select a node among all nodes of the tree, except the root node. For Insert Node, Delete Node, Relabel Node, and Invert Node, the probability of selecting a node is inversely proportional to its number of descendants. Therefore, these transformations are mostly applied on nodes at lower nesting levels. We believe that this strategy reasonably simulates real-world scenarios and is in accordance with the intuition around the concept of duplicates. Structural deviations on nodes at lower nesting levels represent more semantic divergence between two XML trees; hence, duplicates exhibiting deviations on such nodes are expected to occur less frequently. For the remaining transformations, i.e., Delete Path, Delete Subtree, and Swap Subtrees, the probability is uniform among the nodes.

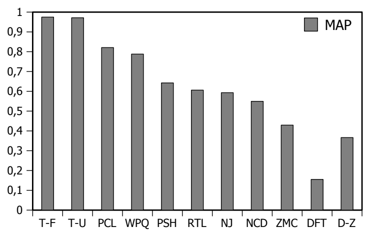
We conducted our experiments on datasets containing 5k trees. Each dataset was generated by first randomly sampling 500 subtrees from the original datasets and then generating 9 fuzzy duplicates per tree. Thus, each generated dataset contains 500 original trees and 4500 duplicates. We also performed the experiments with datasets of different sizes and different distribution of duplicates and the trends were identical. As query workload, we randomly sampled 100 subtrees from the generated dataset and calcuIed the evaluation measures described in Section 2.4.2. Notice that the query workload may contain fuzzy duplicates as well as original, "clean", XML trees.

3.3.4 Comparison of All Similarity Functions

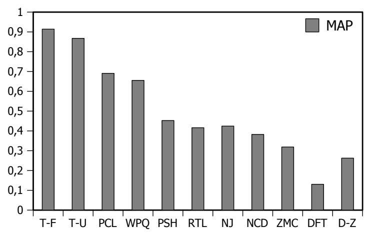
In our first experiment, we empirically study the effectiveness of all similarity functions, for ordered and unordered trees, on the same group of datasets. We aim at obtaining a global comparison between the similarity functions investigated, in particular, comparing our PCI-based approach performs not only to the token-based competitors addressing unordered trees, but also to measures designed to ordered trees such as the TED variants and the techniques proposed in the context of the CDI problem. In order to avoid "apples and oranges" comparisons, we did not apply node-swapping transformations when generating the datasets; hence, the transformations applied did not alter the sibling order, i.e., the original ordering of the trees is preserved in the generated datasets. As a result, the similarity functions for ordered trees are not charged by node permutations between fuzzy duplicates; instead, they only have to disclosure the same set of structural deviations as the similarity functions for unordered trees to identify the duplicates. Thus, our comparison is fair. Nevertheless, we will study the effectiveness of the similarity functions for ordered trees on datasets exhibiting node permutations later in this section. Here, we only



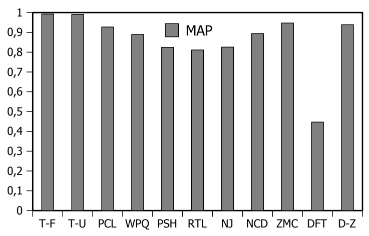
(a) Nasa, low error



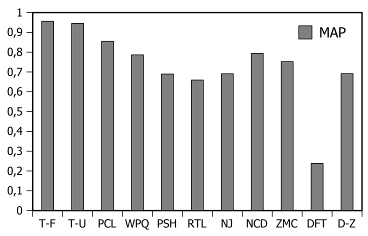
(b) Nasa, moderate error



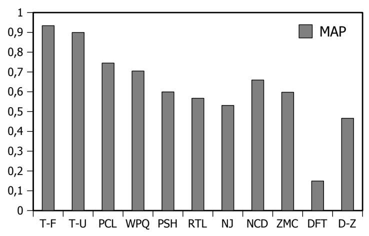
(c) Nasa, dirty error



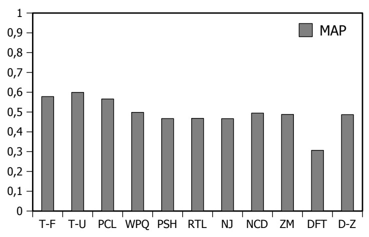
(d) SwissProt, low error



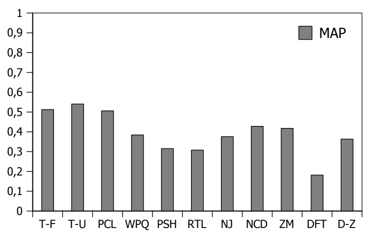
(e) SwissProt, moderate error



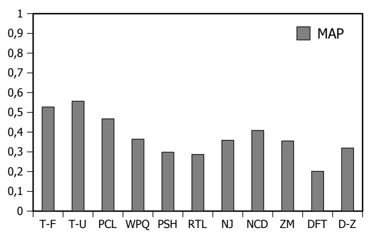
(f) SwissProt, dirty error



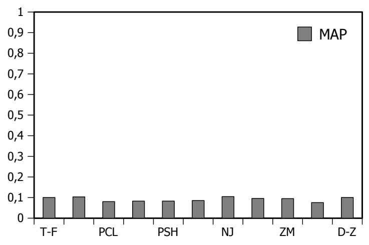
(g) PSD, low error



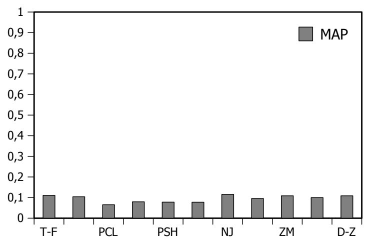
(h) PSD, moderate error



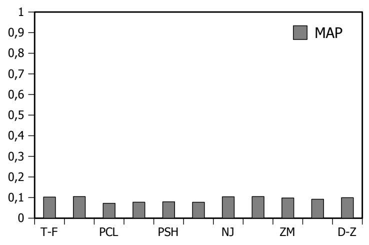
(i) PSD, dirty error



(j) DBLP, low error



(k) DBLP, moderate error



(l) DBLP, dirty error

Figure 3.6: Accuracy results of all structural similarity functions on unordered trees

excluded from our experimental charts the results of the PQ similarity function as it is subsumed by WPQ on unordered trees. Finally, we observed identical trends for all measures described in Section 2.4.2. Therefore, for brevity, we only report the results of the MAP measure.

Figure 3.6 shows the results. We first discuss the results along the different datasets before focus on the comparison between the similarity functions. The effectiveness results on Nasa (Figures 3.6(a)–(c)) and SwissProt (Figures 3.6(d)–(f)) are markedly better than those on the other two datasets. This is expected because these datasets have the largest, richest, and most heterogeneous structure, therefore providing a good structural "signature" to identify XML trees. For low error datasets, almost all similarity functions yielded very good results (MAP value above 0.8) and the best performing ones exhibit practically perfect accuracy. These results confirm that we can identify duplicates by solely relying on structural information.

The results noticeably degrade as the error extent increases. An interesting observation drawn from the experiments is that accuracy on SwissProt is less affected by increases in the level of "dirtiness". As already mentioned, cardinality is a more salient structural information in Nasa, whereas path and label heterogeneity is more prominent in SwissProt. Hence, the results suggest that path and label heterogeneity are more effectively exploited by the similarity functions under study to separate duplicates from non-duplicates; this observation comes as no surprise for the approaches proposed for CDI as such techniques deliberately underweight cardinality differences between trees.

Results are significantly poorer on the PSD dataset (Figures 3.6(g)–(i)). Because the underlying structure is simpler and more homogeneous, MAP values are roughly cut by half for all similarity functions. At the extreme, structural similarity assessment was totally ineffective to identify the duplicates in DBLP (Figures 3.6(j)–l). Because the trees in DBLP are small and have very similar shape, there is very little information to distinguish a tree from one another. Although these results may suggest that structural similarity is useless on datasets exhibiting the same characteristics as DBLP, as we advocated at the beginning of this chapter, approximate matching on structure is nevertheless necessary for EM applications regardless of its discriminative power; for example, it is useful to identify pieces of textual information that represented under different hierarchical nestings. Nevertheless, we will omit the results on the DBLP dataset in the following experimental charts.

Now, we discuss the comparative effectiveness of the similarity functions. All in all, TED-F and TED-U are the best performing similarity functions: they show superior accuracy results and exhibits comparatively little degradation as error extent increases. Unfortunately, TED computation is notoriously expensive to calculate; its runtime is typically orders of magnitude worse than token-based functions (e.g., [ABG10]). DTF is by far the worst performing similarity function: it is ineffective in identifying duplicates even in low error datasets. Moreover, the combination of DFT and ZMC, D-Z, showed poorer performance than ZMC in isolation.

Apart from TED-F and TED-U, PCL is the most effective similarity function in all settings. Moreover, PCL shows less accuracy degradation as the error extent increases as compared to the other token-based and CDI measures. The superiority of PCL against the other token-based similarity functions corroborates our claim that accurate path matching is of paramount importance for EM. In fact, the results show that effectiveness decreases as token granularity w.r.t hierarchical nesting increases: WPQ, which encodes partial paths in the stem part of *wpq*-grams, performed better than PSH and RTL — we used stem of size 2, i.e., $p = 2$—; in turn, PSH, which employs path shingles, fares better than RTL, which uses root-to-leaf paths as tokens. In particular, the performance of the two latter similarity functions drops significantly as the error extent increases. To support even more our conclusion, we ran additional experiments on datasets generated by only one kind of structural modification. We observed that results favoring PCL are, in general, more pronounced on datasets generated by node-level modifications, i.e., node insertion, rename, inversion, and deletion. All these transformations "destroy" path information, thereby requiring approximate path matching, which is more accurately delivered by our PCI-based approach.

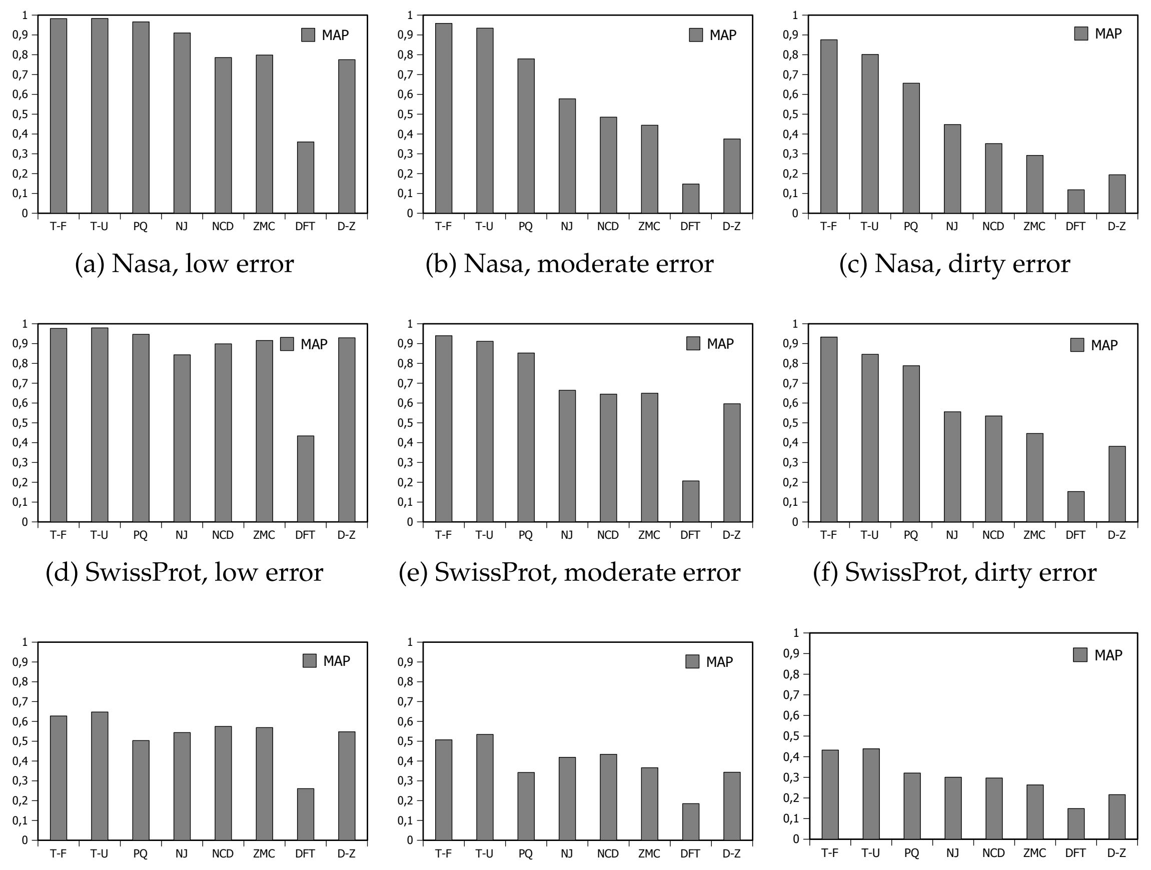
The results of PCL are also superior to those of all CDI similarity functions. While the TED variant of Nierman and Jagadish (NJ) and the entropy-based approaches (NCD and ZMC) are close to PCL on low error datasets, their accuracy severely degrades as the error level increases. As these functions are less sensitive to divergences in the number of occurrences of subtrees, the similarity result between non-duplicates is increased. At higher error extent, many trees are reported closer to other trees than to their duplicates by the CDI functions.

3.3.5 Comparison of Similarity Functions for Ordered Trees

We now focus on similarity functions for ordered trees only. To this end, we used datasets generated by the complete set of transformations, i.e., including the Swap Trees transformation. Also, we now evalaute the PQ similarity function instead of WPQ. The results are shown in Figure 3.7. In general, accuracy is slightly worse in this experiment owing to the sibling permutation between duplicates. Nonetheless, the trends are identical: much better results on Nasa (Figures 3.7(a)–(c)) and SwissProt (Figures 3.7(d)–(f)) as compared to PSD (Figures 3.7(g)–(i)); similarity functions exhibit better robustness on the SwissProt dataset; TED-U and TED-F are the best and DFT is the worst; and, finally, accuracy of CDI functions drops dramatically as error extent increases.

PQ delivered better results than all CDI functions on Nasa and SwissProt. On PSD, however, PQ is worse than NJ for all error extent levels. Accordingly, TED-F is outperformed by TED-U on these datasets—recall that PQ approximates TED-F. This results indicates that, similarly to text, the comparative effectiveness of structural similarity functions can vary across different datasets [CRF03, SB02, CGK06]. Precisely identifying the structural characteristics leading to poorer results of functions such as TED-F and PQ is a very interesting topic for future research.

Among the CDI functions, NJ showed the best results in almost all settings. Regarding the entropy-based approaches, ZMC and NCD delivered very similar accuracy results. In the original paper [Hel07], ZMC was found superior than NCD. As already mentioned, the author conjectured that this result might have been due to the character-level granularity employed by NCD. Our results support this conjecture, as we have increased the granularity of NCD by encoding node labels as numerical values.



(a) Nasa, low error (b) Nasa, moderate error (c) Nasa, dirty error

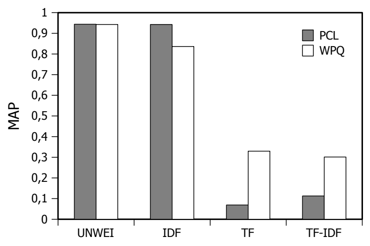
(d) SwissProt, low error (e) SwissProt, moderate error (f) SwissProt, dirty error

(g) PSD, low error (h) PSD, moderate error (i) PSD, dirty error

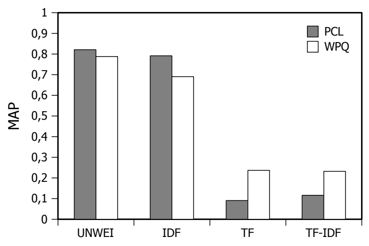
Figure 3.7: Accuracy results of structural similarity functions for ordered trees

3.3.6 Comparison of Similarity Functions for Unordered Trees and Weighting Schemes

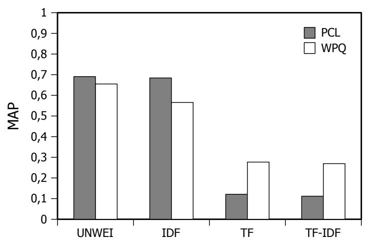
We now focus on similarity functions for unordered trees. We first provide a closer look on the results of Section 3.3.4; here, we only consider PCL and WPQ as they showed the best results. Then, we compare different weighting strategies for structural tokens, namely: unweighted (*UNWEI*), IDF, TF, and TF-IDF (recall Section 2.3.2). For UNWEI and IDF, we used WJS. TF and TF-IDF are based on local statistics (term frequency) and therefore, token weights can vary among different sets. Using *minimum* as aggregation function for Jaccard ensures that similarity results are in the interval $[0, 1]$. However, we experienced unstable results with this formulation. Hence, we refrained from using Jaccard and, instead, employed the well-known Cosine similarity with TF and TF-IDF weighting schemes [SM83]. We also tested the Cosine version of IDF [HCKS08]; however, we preferred to use the Jaccard formulation as we consistently observed superior results with it.



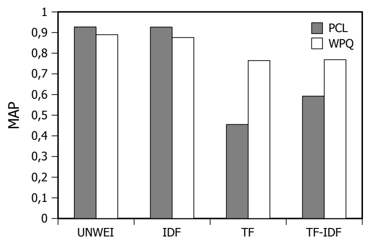
(a) Nasa, low error



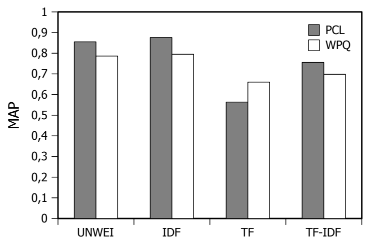
(b) Nasa, moderate error



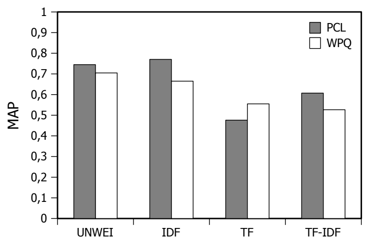
(c) Nasa, dirty error



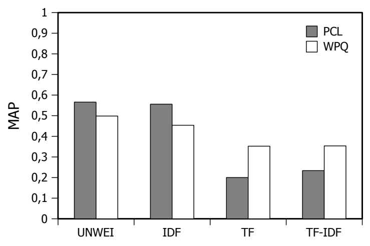
(d) SwissProt, low error



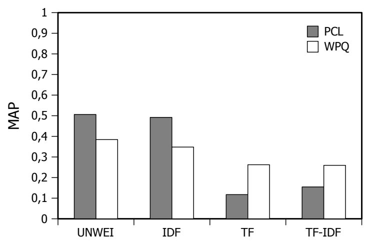
(e) Swissprot, moderate error



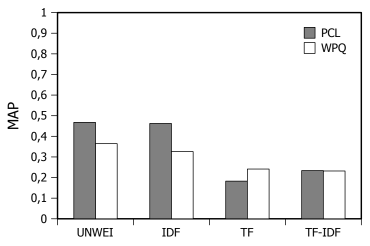
(f) Swissprot, dirty error



(g) PSD, low error



(h) PSD, moderate error



(i) PSD, dirty error

Figure 3.8: Accuracy results for unordered trees using different weighting schemes

Figure 3.8 shows the results. PCL outperforms WPQ in all settings. Especially on SwissProt (Figures 3.8d–(3.8(f)), the accuracy gap between PCL and WPQ noticeably increases with error extent. The better results of PCL are even more impressive when we analyze the average profile size delivered by each representation: for all datasets, PCL produces much more compact profiles, about 4–5x shorter than WPQ. To capture subtree permutations, WPQ has to produce an increased number of *wpq*-grams—depending on node's fanout—, while for PCL, the number of tokens is determined by the number of paths.

We now analyze the results of the comparison of weighting schemes. The first observation is that UNWEI and IDF perform consistently better than TF and TF-IDF. This result might well be due to the use of Jaccard for UNWEI and IDF. When comparing the unweighted and IDF-weighted versions of Jaccard, the results are inconclusive: for PCL, IDF is slightly better on SwissProt, while being marginally outperformed by UNWEI on Nasa; on PSD, the results of both strategies are practically identical.

3.3.7 Experimental Summary

For datasets containing trees with rich structural information, it is possible to accurately identify duplicates by solely relying on structural similarity functions: most similarity functions investigated were effective on Nasa and SwissProt, especially on datasets with low error extent. In this vein, we found similarity assessment, in general, more resilient to increases of error extent on SwissProt, which exhibits path and label heterogeneity more prominently, than Nasa, which is characterized by dispersed cardinality distribution (i.e., distribution of the number of nodes and paths per tree). Accuracy results on PSD are considerably poorer; on DBLP, structural similarity was found completely ineffective in identifying duplicates.

TED-F and TED-U showed the best accuracy results overall. TED-F outperformed TED-U on Nasa and SwissProt, whereas, on PSD, TED-U achieved better results. Similar trend was observed on PQ, which approximates TED-F: PQ outperformed all CDI similarity functions on Nasa and SwissProt, but showed worse accuracy than NJ on PSD. NJ was the best performing among the similarity functions designed for the CID problem; the results of NCD and ZMC were similar. DFT presented very poor performance in all datasets and the combination between ZMC and DFT did not improve the results of the former. In general, the accuracy of all similarity functions proposed in the CDI context degrades more sharply as error extent increases.

For token-based similarity functions, we found the Jaccard similarity being superior to Cosine. Also, we did not observe any significant gain by using weighting schemes.

For unordered trees, PCL is clearly the structural similarity function of choice: it provides superior accuracy results, its performance degrades less steeply as the dataset level of “dirtiness” increases, and produces much shorter profiles. Comparatively, PCL also provides superior results as compared to the CID similarity functions.

3.4 Summary

In this chapter, we presented a comprehensive study on structural similarity functions in the EM context. We started by reviewing on a highly representative set of structural similarity measures for ordered and unordered trees. We then proposed a novel method for producing high-quality structural representations of unordered trees. In this context, we exploited structural summaries and clustering concepts to represent XML documents as profiles, where each token embodies a set of similar structural patterns. Remarkably, our approach delivered substantially more compact token sets than competing approaches, while providing more accurate results. In this context, we also explored different ways to weigh structural XML representations.

Chapter 4

Combining Text and Structure

The accuracy of classification tasks can be substantially improved by combining multiple classification models together instead of using a single model in isolation. This approach, which we generally refer to as combination of evidence, has been extensively adopted in several application domains including data mining, machine learning, and pattern recognition [TSK06, Bis06][1]. In Chapter 2, we have seen that EM and IR tasks have a natural interpretation as a classification model and, very often, combine evidence obtained from multiple sources, such as comparison fields, similarity functions, and document representations. Regarding XML similarity, combination of evidence is not only useful for improving accuracy, but is also at the heart of the very notion of similarity itself owing to the intrinsic duality of structure and text in XML.

In the previous chapter, we have shown that structural similarity alone can sometimes provide highly accurate EM results. Nevertheless, most items of identifying information of real-world XML data, which allow discriminating trees in a collection from each another, are represented by text nodes. Because we have to anticipate the same sorts of data quality issues found in the relational world, e.g., typos and misspellings, similarity matching has to be applied on text data as well. Hence, accurate and robust similarity functions for XML data should address both text and structure and combine their similarity results in a meaningful way.

The interplay between text and structure also appears in other aspects of the similarity assessment. When multiple text nodes are used as comparison fields, structure is necessary to avoid comparison between unrelated text data, e.g., *name* vs. *address*. In this regard, instead of concatenating text nodes values and treating them as a larger string, a better approach is to contextualize each textual unit (i.e., textual token) with structural patterns. Furthermore, such text contextualization is important to prevent statistical distortion when a string appears in unrelated text nodes but with different frequencies. This distortion would jeopardize weighting schemes based on statistical information such as IDF. For example, if "Alice" is very common within text nodes related to *name* but quite rare within *address* nodes, then all the tokens generated from

[1]In these areas, the combination of classification (or regression) models is commonly referred to as *ensemble methods*.

"Alice" will however be assigned a low weight in the concatenated name-address string. Note that the motivation for contextualizing text nodes in XML data is the same as for applying text segmentation on flat strings in relational data.

Further, it is often the case that only part of the available textual information is interesting for similarity matching, i.e., simply using all textual information available will hardly be effective. Similar to the selection of comparison fields in relational data for Classification Model Design (recall discussion in Section 2.1.2), issues such as lack of independence are known to negatively affect the quality of EM results when more fields than needed are used. Also, XML data in particular may contain very long strings — especially document-centric XML, which are typically inappropriate for identify duplicates. Finally, adding long strings to the tree representation would blow up the size of token sets thereby hurting performance (see Chapter 5). To tackle this problem, we adopt the strategy of exploiting structural information together with user-defined textual information; these pieces of information compose the entity description of a tree. In this context, we incorporate into our framework sub-queries in the form of simple structural constraints specifying which text nodes will form the textual part of an entity description. We call such sub-queries *entity description selection queries* (EDS queries, for short). The remaining elements, i.e., those not selected by the EDS query, form the structural part of the entity description. In this context, we also aim at supporting exploratory interaction and user guidance for formulating EDS queries.

The formulation and evaluation of EDS queries is considerably more complicated in heterogeneous XML datasets. The underlying structure is frequently very complex and the user may have only limited knowledge about it, which render schema-aware query languages like XQuery [BCF+07] or XPath [BBC+07] inappropriate. Even if the structure is known, the complexity would however be reflected in the queries leading to clumsy and error-prone query formulations. To overcome this drawback, we follow the flexible query model often adopted in XML search [AYL06]: structural constraints are interpreted *approximately*, i.e., it is not required that the constraints expressed in the queries exactly match patterns in data instances. As for structural similarity, we have the need of approximate path matching where prior identification of duplicate paths can be exploited to support EDS queries.

There are several challenges in incorporating text and structure into a single notion of similarity and, in turn, into our similarity join framework. Textual and structural patterns have different semantics and usually exhibit very variable frequency distributions across different datasets. This fact exacerbates the combination of evidence problem, in particular, for approaches that adopt a linear combination of similarity scores (see discussion in Chapter 1). Also, the experimental results in Chapter 3 revealed that, while being effective in identifying duplicates for some datasets, structural similarity may perform very poorly on datasets showing regular structure. On the other side, because entity description selection is done approximately, textual similarity may be negatively affected if text nodes are erroneously selected for similarity evaluation. Therefore, the combination strategy must be robust to the situation in which either textual or structural similarity does not properly separate duplicates from non-duplicates.

From the viewpoint of efficiency, the adoption of token-based similarity functions is of paramount importance for combining textual and structural similarity. Besides the wealth of optimization opportunities (see Chapter 5), token-based similarity functions allow measuring textual and structural similarity between XML trees by operating on token sets representing text, structure, or both, in a unified framework. Thus, we can obviate multiple accesses to XML trees and separate invocations to subroutines or operators for calculating structural and textual similarity (see discussion in an XDBMS environment in Chapter 6). Furthermore, as we will see shortly, token-based methods facilitate the integration of approximate entity description selection with the similarity join processing and the assignment of a clean semantics for the separation of structural and textual entity descriptions.

Our general approach for the combination of text and structure embraces two key ideas: decomposition of the set of path clusters of an XML collection into structural and textual subsets and generation of tokens that jointly capture textual and structural information. For the former, we exploit the results of EDS queries. For the latter, we propose prepending q-gram tokens with structural information. By doing so, we are able to both avoid the comparison of unrelated concepts and keep fine-granular token statistics. As in Chapter 3, we consider combination methods for ordered and unordered trees. For ordered trees, we introduce the concept of *extended pq-grams* that embeds q-grams into pq-gram tokens; we propose and analyze three token generation strategies for this new concept. Combination methods for unordered trees are based on the PCI technique introduced in Chapter 3; in this context, we explore approaches to combination of evidence at the token and score levels. We also exploit PCIs to devise a compact structure to approximately match user-specified structural constraints for entity description selection. We call this structure, which approximately represents all path clusters of an XML collection, *Path Cluster Summary* (PCS). We realize PCS as little memory-resident inverted lists providing efficient evaluation of queries for entity description selection. Moreover, PCS can be used to update clustering information and avoid the need of repeating the clustering process when new documents enter datasets under consideration.

The rest of this chapter is structured as follows. In Section 4.1, we formally define the decomposition of the path cluster set. In Section 4.2, we introduce the *epq*-gram concept for ordered trees. PCI-based approaches for combination of evidence on unordered trees are described in Section 4.3. The PCS structure is presented in Section 4.4. We show the results of our empirical experiments in Section 4.5. Related work is discussed in Section 4.6, before we wrap up with the summary in Section 4.7.

4.1 Text and Structure Delimitation

Recall from Chapter 3, the result of the PCI generation process over the PS of a collection of XML documents is the set of clusters PC_{θ}^{tgl}. We shall decompose PC_{θ}^{tgl} into two disjoint subsets, which are used to delimit the structural and textual part of entity descriptions. We now formally define these subsets; details about how this decomposition is realized using EDS queries are given in Section 4.4. (In the following, we omit the target label and the threshold cutting point in the notation for a set of path clusters to avoid clutter, i.e., $PC^{tgl}_{\theta} = PC$.)

Definition 4.1. *Let C be a collection of XML documents; $PC = \{pc^1, \ldots, pc^n\}$ is a set of clusters generated from the PS of C. We decompose PC into two disjoint sets, PC_s and PC_t, such that $PC_s \cup PC_t = PC$. (PC_t and PC_s do not form a partition, however, because one or the other set can be empty.) We call PC_s the set of structural path clusters and PC_t the set of textual path clusters. Further, given a text node u, we say that $u \in PC_t$ if u appears at the end of some path p, s.t., $p \in pc^i \wedge pc \in PC_t$.*

The following tokenization functions we present use the PC_t set (received as a free parameter) to guide the generation of tokens containing textual information. For simplicity, we omit the PC_t from the list of parameters.

4.2 Combination Approach to Ordered Trees

In this section, we study ways to simultaneously deal with structural and textual similarity on ordered trees. Tokenization methods for ordered trees must embody sibling relationship information. In the previous chapter, we reviewed the *pq*-gram tokenization method, which captures this information. A different tokenization method for ordered trees was presented by Yang et al. [YKT05]. The authors convert a tree into a full binary tree representation. The token set consists of all subtrees composed by a node of the original tree and its two children (such tokens are referred to as *binary branch* by the authors). *pq*-grams provide more flexibility regarding the subtree shape, which can be used to control structural sensitivity. Moveover, as we have seen in the last chapter, the PQ similarity delivers quite accurate results. Therefore, we build upon the *pq*-gram technique to construct tokens containing both structural and textual information.

We propose an extension of the original definition of *pq*-grams. During generation of the *pq*-gram profile, we convert text nodes to sets of *q*-grams, which are used to enrich *pq*-grams with textual information, leading to *extended pq-grams* (*epq*-grams, for short) [RH08a]. For the *epq*-gram generation, we focus on the conceptual representation of strings in an expanded tree, denoted by $ET^{p,q}$. This approach allows us to use an almost identical algorithm to produce *epq*-grams from a stream of nodes, with minor variations to handle text nodes. Furthermore, the generated grams seamlessly reduce to normal *pq*-grams when the input stream only contains structural nodes. There are several conceivable ways to represent strings as nodes in an expanded tree. Next, we analyze three versions.

The first alternative consists of considering each character of a string as a *character node*. Hence, whenever a parent of a text node is selected as an anchor node, q character nodes are selected to form a new *epq*-gram version called *epq*-v1. Given a $T^{2,2}$

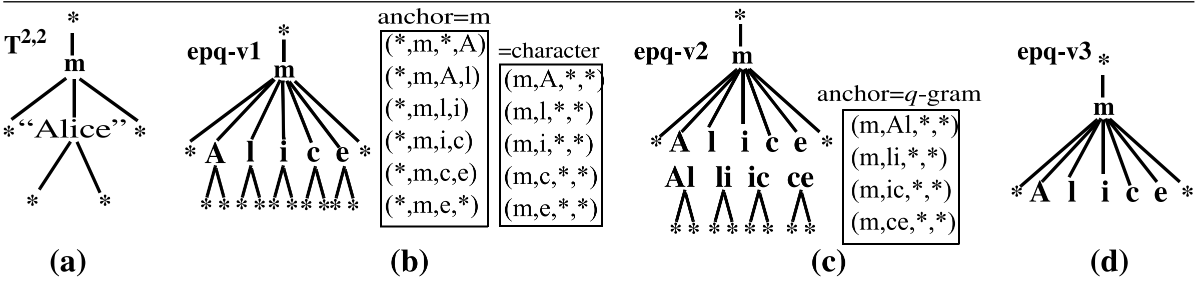


Figure 4.1: Versions of extended *pq*-grams

in Figure 4.1(a), the corresponding $ET^{2,2}$ for *epq-v1* together with the resulting *epq*-profile are shown in Fig 4.1(b). Note that the *epq*-profile is separated into two subsets: *epq*-grams having *a* as anchor node, the other having character nodes as anchor node. When the node labels are concatenated, sequences of character nodes form q-grams, which are combined with structural information. Unfortunately, *epq*-v1 always forms *1*-grams when the character node is the anchor, which is independent of the choice of q — e.g., the 1-gram (highlighted) in the middle of the token *m**A***** . Note that for $q = 1$, different strings containing an identical (*multi-*) set of characters have the same q-gram profile and, hence, maximum similarity (see Section 2.3.2).

To prevent the potential drawback of *epq*-v1, we propose a hybrid approach, called *epq*-v2 (see Figure 4.1(c)). Now, character nodes are used when the parent is the anchor node, and *q-gram nodes* when the text node itself is the anchor. As a result, all *epq*-grams with textual information now contain q-grams of the same size (*epq*-grams having *a* as anchor node are the same to those of *epq*-v1). Note that *epq*-v2 requires an additional parameter specifying the q-gram size. Here, we use the same value of q for both *pq*-grams and q-grams; they can be independently chosen, though.

The previous versions may consume substantial space because of large profile sizes. This observation motivates the third approach, *epq-v3*, which is derived by using character nodes, but pruning their *q-null* children from the expanded tree (see Figure 4.1(d)). Compared to the previous versions, this approach roughly produces only half of the *epq*-grams embodying text; therefore, textual similarity receives less weight. However, this property can be compensated by the fact that tokens containing text are likely to be less frequent than structure-only tokens. The rationale is that by using common notions of weights that are inversely proportional to frequency, e.g., IDF, we can balance the effect of the q-gram reduction. In Section 4.5, we empirically evaluate this conjecture.

We denote by $epq[v\{i\}]$[2] the tokenization function that generates epq-v{i} tokens, for $i = \{1, 2, 3\}$; The corresponding similarity function is denoted by $EPQ_{v\{i\}}$. Theorem 4.1 shows the relation between the resulting profile cardinality of *epq*-v1 and the numbers of non-leaf nodes, empty nodes, and text nodes.

[2]Parameters p and q are omitted.

Table 4.1: epq-gram profile cardinalities

Version	$\lvert epq[v\{i\}](T)\rvert$
epq-v1	$kq + 2e + 2tn - 1$
epq-v2	$kq + 2e + t(2n - q + 1)$
epq-v3	$kq + 2e + tn - 1$

Theorem 4.1. *Let $p > 0$, $q > 0$, and T be a tree with e empty nodes, k non-leaf nodes and t text nodes. Assume that all text nodes have a fixed length of n. The size of the extended epq-gram profile (version 1) is: $|epq[v\{i\}](T)| = kq + 2e + 2tn - 1$.*

Proof (Sketch)**.** *Theorem 4.1 can be shown by structural induction similarly to the strategy used in [ABG05]. The deletion of leaves should be done in two stages: first deletion of empty nodes and then deletion of text nodes. Deleting a text node decreases the cardinality of the pq-gram profile by $2n$ if the text node has siblings, otherwise by $2n + q - 2$; deletion of an empty node decreases the cardinality by q if the node has no siblings, otherwise by 2* □.

The profile cardinality of *epq*-v2 and *epq*-v3 can be derived similarly. Table 4.1 shows the cardinality of the three *epq*-gram versions. Note that, especially when applied to text nodes with long strings, the *epq*-gram profile can have a cardinality considerably larger than that of normal *pq*-grams. Therefore, we define the expanded tree $ET^{p,q}$ of a tree T to contain only text nodes in PC_t.

4.3 Combination Approach to Unordered Trees

The combination of text and structure for unordered trees is based on the *pcl* tokenization function. Following the decomposition of the set of path clusters PC, we parameterize *pcl* with *s* or *t* indicating the generation of structural-only tokens or tokens containing textual information, respectively; The profile returned by $pcl[s]$ corresponds straightforwardly to the set of PCIs relative to PC_s. Next, we describe the *pcl*[t] tokenization function and, then, discuss strategies for using *pcl*[s] and *pcl*[t] to obtain a single similarity function.

4.3.1 Textual Tokenization Function

In the following, we first define the *pcq* tokenization function, which generates tokens combining path cluster information and *q*-grams — called *pcq*-gram tokens — and, then, the $pcl[t]$ tokenization function, which uses *pcq* as auxiliary function.

Definition 4.2 (*pcq* Tokenization Function)**.** *Given a tree T, let u be a text node of T appearing under a path cluster pc^i. Let $qgram(u)$ be the qgram profile of the content of u as defined in Section 2.3.2. The pcq tokenization function generates a profile from the u as follows:*

$$pcq(u) = i \circ q : u \in pc^i \wedge q \in qgram(u) \; .$$

Definition 4.3 (pcl[s] Tokenization Function)**.** *The pcl[s] tokenization function generates a profile from a tree T as follows:*

$$pcl[s](T) = \bigcup_{u \in C(T) \wedge u \in PC_t} pcq(u) \; .$$

Note that, besides being used to compose the textual representation of a document, PCIs also serves to approximately locate text nodes. For example, in Figure 2.4, we are able to compare the text value of `mother` in subtrees $a)$ and $b)$ because their respective paths have associated the same $PCI = 4$ (see Figure 3.3). The collection statistics of a *pcq*-gram token t is constrained at the path cluster pc^i, i.e., $freq(t, C)$ corresponds to the number of documents where token t appears in a collection associated with a path $p \in pc^i$.

4.3.2 Combination Strategies

We now consider ways to combine textual and structural similarity for unordered trees. We examine two approaches: score-level combination (*SLC*) and token-level combination (*TLC*). The first is the linear combination of the similarity scores derived from $pcl[s]$ and $pcl[t]$ — the corresponding similarity functions are denoted by PCL^s and PCL^t, respectively. In this approach, we have two different profiles for each tree, the similarity functions are evaluated independently, possibly using different weighting schemes, and their result is combined using weights that can be either hand-tuned or obtained from some learning model. Note that PCI_t could be further segmented into subsets corresponding to text nodes related to different concepts (e.g., different textual profiles for `name` and `address`) with weights associated accordingly. The combination of several comparison fields is often employed on relational data as discussed in Chapter 2. Nonetheless, in this thesis, we focus on the combination of text and structure and, hence, we will use a single textual profile for each subtree.

The second approach builds the union of the structural and textual profile of a document thereby obtaining a unified representation. In other words, we generate one profile per tree containing structural tokens and tokens embodying textual and structural information. We can employ a weighting scheme such as IDF to assign weights to tokens based on their statistical properties. In this context, tokens containing textual information would usually have lower frequencies than structural tokens and, therefore, larger *IDF* weights. Finally, note that the combination method adopted for *epq*-grams is equivalent to *TLC*. In the following, we formally define PCL_{slc} and PCL_{tlc}, the similarity functions obtained from score- and token-level combination strategies, respectively.

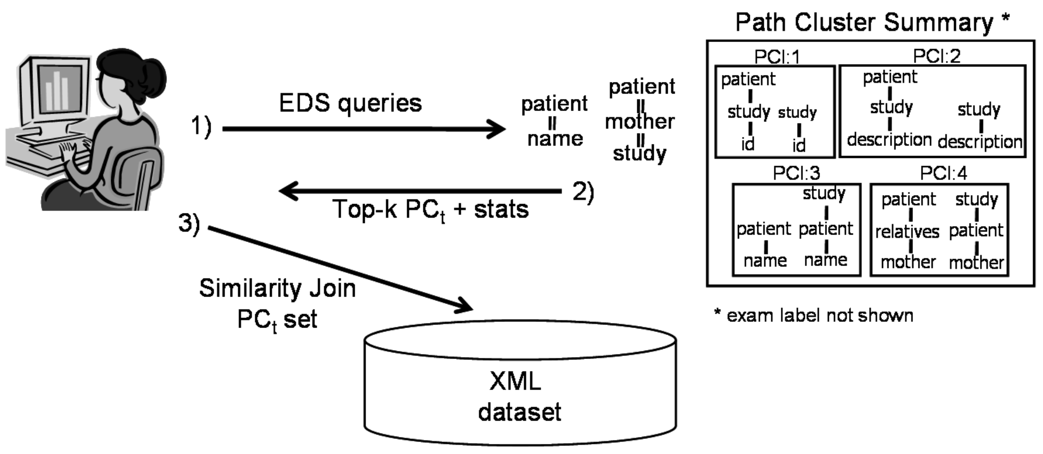


Figure 4.2: EDS queries on PCS

Definition 4.4 (PCL_{slc} Similarity Function). *Let T_1 and T_2 be two XML trees. Let λ_t and λ_s be similarity weights such that $\lambda_t + \lambda_s = 1$. The score-level similarity combination (PCL_{slc}) between T_1 and T_2 is given by:*

$$PCL_{slc}(T_1, T_2) = \lambda_s \times PCL_s(T_1, T_2) + \lambda_t \times PCL_t(T_1, T_2) \tag{4.1}$$

Definition 4.5 (PCL_{tlc} Similarity Function). *Let T_1 be an XML tree. The combined profile of T_1, denoted as pcl[s,t], is given by $pcl[s,t](T_1) = pcl[s](T_1) \cup pcl[t](T_1)$. Similarly, consider an XML tree T_2. We denote by PCL_{tlc} the similarity function defined by $\langle pcl[s,t], \_, \_\rangle$.*

4.4 Entity Description Selection Using a Path Cluster Summary

So far, we have described how we accomplish XML path clustering (Chapter 3) and how we compute textual and structural similarity. We are now ready to discuss the process of determining the sets PC_t and PC_s. Because this decomposition of the original set of clusters defines which parts of an XML tree will deliver textual or structural tokens, it is a key design decision. Further, the definition of the best decomposition configuration is critically dependent on the application domain. In fact, as discussed in Section 2.1.2, Classification Model Design invariably requires human interaction. Hence, we let users specify the PC_t set by issuing an EDS query. This query consists of a set of simple path specifications, such as `/a/b/c`,[3] that are approximately matched against the set of clusters. The top-K answers for each path specification, i.e., the K cluster sets with highest similarity scores, are selected to constitute PC_t;

[3]Note that XPath-style filtering predicates, e.g., `/a[b]/c`, actually defines a *twig* pattern and are currently not supported.

the remaining cluster sets form PC_s. Besides the parameter K, a threshold τ_{eds} can be specified to define a minimum similarity value between a path specification and cluster prototypes.

Further, because users are likely to have only vague knowledge about the underlying structure of the data collection, we support exploratory queries for cluster decomposition, before the actual similarity join evaluation takes place. Currently, we return several pieces of statistical information about the top-*k* results, such as cluster frequency in the collection and information about string length of text nodes appearing in the cluster set. The user can therefore reformulate the query if, for example, the returned cluster set appears only in a few subtrees or is related to a part of the document with predominance of lengthy free text. We revisit this aspect in Chapter 6.

Definition 4.6 (Entity Description Selection Query (EDS Query)). *Let PC be a set of path clusters, P a set of distinct path specifications, K a positive integer, τ_{eds} a constant threshold in the interval $[0,1]$, and stats a Boolean value. An EDS query is then given by:*

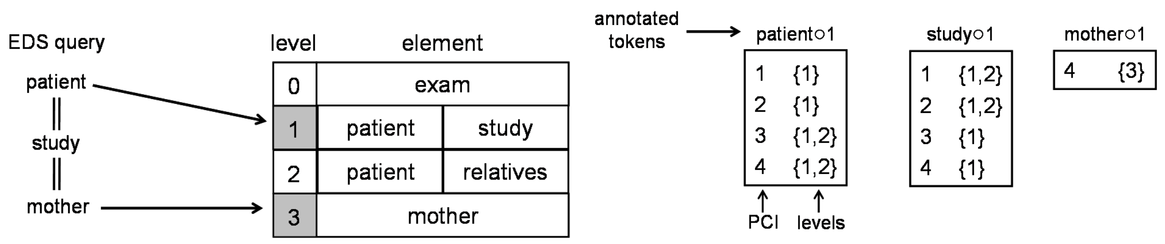
$$EDS(P, K, \tau_{eds}, stats) = \bigcup_{p \in P} eds(PC, p, K, \tau_{eds}, stats) \, , \tag{4.2}$$

where eds is a function that compares p with all path clusters in PC and returns the k most similar path clusters represented by the respective PCIs, for $k \leq K$, whose similarity score with p is not less than τ_{eds}; if the value of stats is True, available statistical information about the k most similar path clusters is included in the result.

The parameter *stats* is set to *True* to enable interactive and exploratory user sessions prior to similarity join execution. In this setting, the user may issue several EDS queries, analyze the statistics returned with the results, and, then, select the set of most interesting path clusters to compose PCI_t before executing the similarity join. Figure 4.2 illustrates this process. On the other hand, setting *stats* to *False* allows embedding the EDS query into the similarity join as sub-query.

Now, we provide implementation details for evaluating EDS queries. To avoid the burden of comparing path queries with all path classes of a cluster, we exploit the so-classed *Path Cluster Summary* (PCS), a structure approximately subsuming all path clusters of an XML collection. Path queries are matched with *cluster prototypes* (aka cluster representatives) rather than with each path class. To this end, we adopt a simple but effective level-based structure in which all (annotated) labels appearing at the same level are kept together. Figure 4.3(a) shows the prototype of the path cluster with *PCI* 4 in our running example. The matching between EDS queries and a prototype is done using the WPS similarity function, similarly to that of regular paths. The main difference is that only a single label match is allowed per level. For example, in Figure 4.3(a), the element `study` of the EDS query cannot be matched with the cluster prototype because of the previous match of `patient` at level 1.

We design a specialized inverted list index to represent the set of cluster prototypes in PCS and efficiently match EDS queries against them. Inverted lists are constructed



(a) Cluster prototype (PCI 4) (b) Inverted list of cluster prototypes

Figure 4.3: Path cluster prototype matching

for each annotated label appearing in a cluster prototype. Figure 4.3(b) shows the inverted lists of the annotated labels appearing in the example of Figure (4.3a). Note that there is no need to create an inverted list for the target label, because it is the root label of all cluster prototypes. Accordingly, we ignore the first appearance of the target label in an EDS query — which yields the annotated token `tgl∘1` — as its match score is uniform along all prototypes.

The space requirement of the index is $O\left(|L| \times |P| \times D\right)$, where L is the set of distinct annotated labels in the dataset — L corresponds to the dictionary of the index and its size can be greater than the number of distinct labels due to recursion —, P is the set of distinct paths, and D is the maximum path length among all documents. Clearly, this analytical bound is "loose". In practice, a label appears mostly in a few paths and levels; thus the space consumption is expected to be much smaller. Moreover, the vast majority of XML datasets present fairly moderate values for $|L|$, $|P|$, and D, e.g., less than 200 path classes and a maximum depth of 8 [HMS07]. Therefore, for such datasets, the dictionary and the inverted lists can be kept memory-resident. Finally, path cluster statistics are stored on separate table that is accessed at the end of the evaluation process for each PCI in the result.

Furthermore, we can ensure truly interactive response times even under concurrent requests by employing well-known IR optimizations such as the *max-score* strategy [TF95].[4] The order of evaluation of the inverted lists follows the order of the EDS query components. This means that the upper bound for the weight of matching labels decreases monotonically as the query is evaluated (recall that we use the minimum as aggregation function in Equation 2.11) and, therefore, it is easy to compute the highest score that a cluster prototype containing all remaining components of the EDS queries can achieve. Thus, we can promptly stop the query evaluation and return the current top-*k* candidate as soon as there is no other candidate in the intermediate result whose highest possible score is higher than the score of the *k*th

[4]Because the evaluation of EDS queries can be interpreted as a *set selection query*, we might well employ (with minor modifications) similarity matching algorithms in an even more sophisticated way, e.g., see [HCKS08]. Nevertheless, the max-score optimization already provides interactive responses on the compact PCS structure.

candidate. In practice, only the inverted lists associated with the first query label components will be evaluated. Note that we are interested only in the top-k results; their complete score is unimportant.

Besides EDS query matching for cluster set decomposition, our inverted list representation of cluster prototypes is also important for (PCI-equipped) PS maintenance in the presence of incremental path class updates. New path classes are matched against the index to automatically select the most similar path cluster. Therefore, the need of complete re-clustering is avoided after changes in the PS structure (e.g., when a new document is stored or an edit operation such as node insertion is applied), or it can be postponed to be done off-line. A pre-defined threshold θ' is used to define a minimum value of closeness between a new path class and cluster prototypes. If no cluster prototype is returned with similarity to the new path class not less than θ', a new cluster is created and the path class is assigned to it. Typically, θ' is defined with the same value of the cutting threshold θ that was used to generated the initial set of path clusters (see Section 3.2.3). Note that the creation of a new cluster also triggers an update on the PCS structure itself. We discuss the maintenance of the PCS structure in Chapter 6.

4.5 Experiments

In this section, we measure the effectiveness of our approaches for text and structure combination on several real-world XML datasets. We also evaluate the benefits of increasing the PC_t set by using more EDS queries.

4.5.1 Approaches Evaluated

We evaluate the following similarity functions:

- $EPQ_{v\{i\}}, i = 1, 2, 3$: defined by $\langle epq[v\{i\}, p = 2, q = 2], Jaccard, IDF\rangle$. In the experimental charts, we abbreviate $EPQ_{v\{i\}}$ by *Vi*.

- PCL_{tlc}: defined by $\langle pcl[s, t, q = 2], Jaccard, IDF\rangle$. In the experimental charts, we abbreviated PCL_{tlc} by *TLC*).

- PCL_{slc}: defined by $\lambda_s \times PCL_s + \lambda_t \times PCL_t$, where PCL_s and PCL_t are, in turn, defined by $\langle pcl[s], Jaccard, IDF\rangle$ and $\langle pcl[t, q = 2], Jaccard, IDF\rangle$, respectively. Note that we use the IDF weighting scheme. Although the use of unweighted structural profiles was shown to perform slightly better as compared to the use of IDF-weighted profiles, we did not observe any accuracy gain by using unweighted structural tokens in our experiments. We evaluate instances of PCL_{slc} with structural (textual) weights varying from 0.1 (0.9) to 0.9 (0.1) and report the best result; in the charts, PCL_{slc} is abbreviated by *SLC*.

Table 4.2: Dataset statistics

Dataset	P_t	Text size mean/min/max
SwissProt-Pt1	`/Entry/Ref/Author`	108.3/4/4670
SwissProt-Pt2	`/Entry/Ref/Author` `/Entry/Org`	202.8/25/4771
Nasa-Pt1	`/dataset/title`	72.7/5/234
Nasa-Pt2	`/dataset/title` `//journal/author/lastName`	98.8/5/632
PSD-Pt1	`/ProteinEntry/sequence`	330.8/3/27374
PSD-Pt2	`/ProteinEntry/sequence` `/ProteinEntry/organism/ formal`	368.4/15/27386
DBLP-Pt1	`/inproceedings/title`	66.9/1/298
DBLP-Pt2	`/inproceedings/title` `/inproceedings/author`	104.5/5/1509

Finally, to better understand the effects of combining structural and textual similarity, we also report the results of the evaluation of PCL_s and PCL_t in isolation (abbreviated in the charts by S and T, respectively).

4.5.2 Datasets

The generation of datasets containing fuzzy duplicates followed a process analogous to that described in Chapter 3. The main difference is that we now maintain text nodes in fuzzy duplicates and, besides structural transformations, also textual transformations are applied to text nodes. Textual transformations consist of character-level operations, namely insertions, deletions, and substitutions. These transformations aim at simulating typical data entry errors, such as misspellings; to this end, we used statistics from empirical studies in spelling correction techniques [Kuk92].

Further, we restricted the set of nodes on which textual transformations are performed to those appearing at pre-defined path classes, which we refer to as P_t. For example, for a dataset derived from DBLP, we applied textual transformations only on text nodes under the path `/inprocedings/title`. Like structural transformations, the percentage of characters affected by the set of textual transformations is defined by the error extent parameter. We used the same value of error extent for both structural and textual transformations and adopt the same categorization employed in Chapter 3, i.e, low (10%), moderate (30%), and dirty (50%) error datasets. Note that error extent refers to the set of nodes eligible for textual transformation collectively and not to each node individually.

Table 4.2 gives details about the textual information of the source datasets. We use the suffix *Pt1* and *Pt2* to indicate datasets generated using one and two path classes, respectively (i.e., $|P_t| = 1$ or $|P_t| = 2$). PSD has the largest textual part, while DBLP has the smallest. The mean text size of Nasa and PSD only increases moderately from Pt1 to Pt2; on the other hand, the size of the textual part of SwissProt-Pt2 is twice as large as that of SwissProt-Pt1.

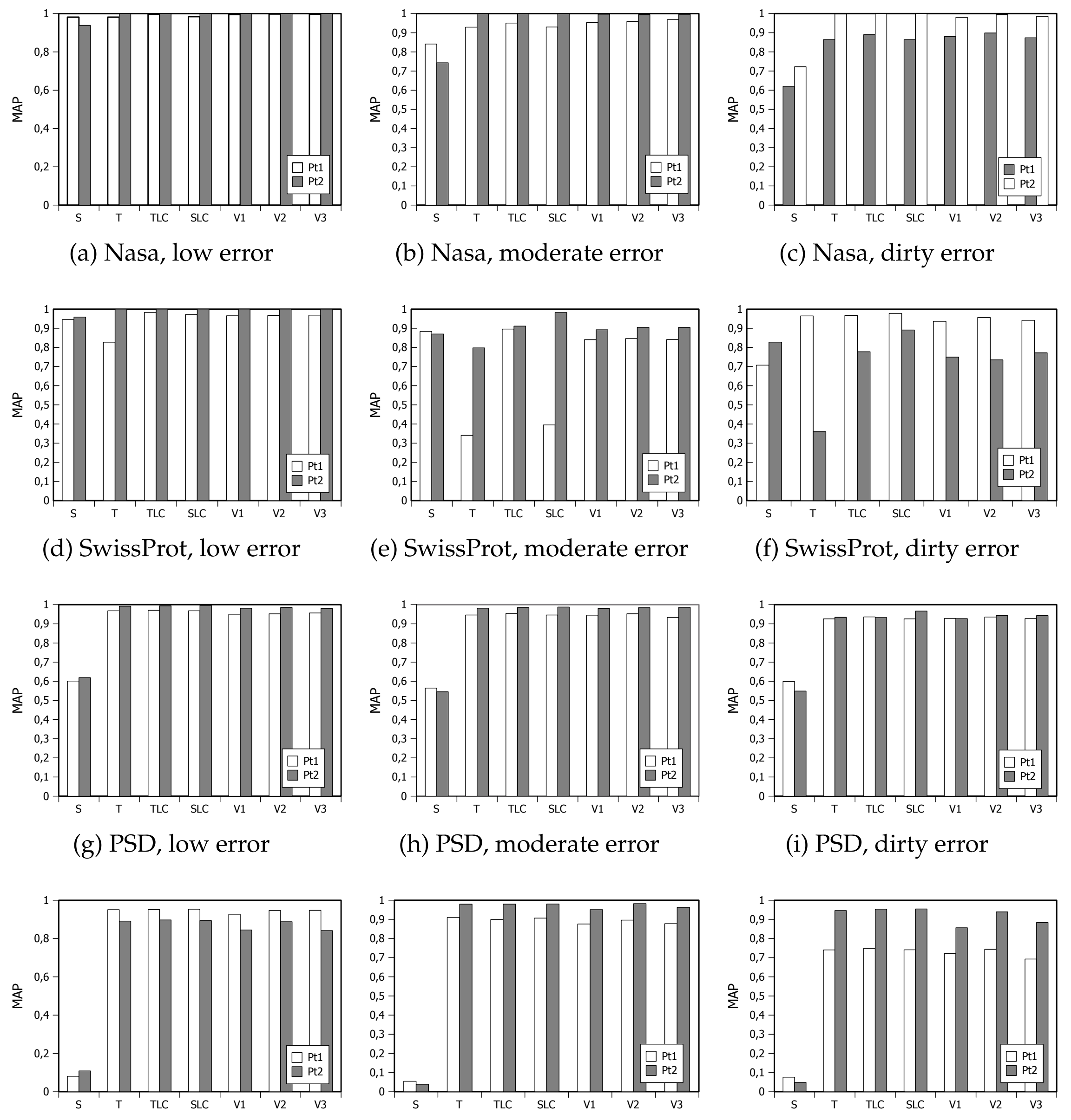
Structural transformations, as described in Chapter 3, are applied after textual transformations. Hence, the path of a text node on whose content textual transformations were applied may be altered by node-level transformations. On the other hand, a node selected for textual transformation cannot be deleted afterwards; note that this restriction prevents some structural operations from being applied on this node (i.e., Delete Node) as well as on its ancestors (i.e., Delete Path and Delete Subtree).

In the experiments, we used EDS queries with $P = P_t$ (see Equation 4.2), i.e., the same path classes specified for generating a dataset were used as path specifications for the EDS query issued against this dataset. Further, we set $K = 1$ and $\tau_{eds} = 0$, which means that, for each path specification, the EDS query returns exactly one PCI. Notice that, owing to the structural transformations and the inherent imprecision of the path cluster process, the PC_t set returned by the EDS query may contain "extraneous" or "spurious" PCIs, i.e., PCIs whose associated paths are, in large part, not in P_t or were not derived therefrom by structural transformations and, thus, contain text data related to different concepts.

4.5.3 Comparison of All Similarity Functions

Similarly to Chapter 3, we compare the similarity functions for ordered and unordered trees on the same group of datasets by leaving out node-swapping transformations when generating fuzzy duplicates. Likewise, we observed identical trends regarding the similarity functions for ordered trees, i.e., the three versions of EPQ, on datasets generated by a set of transformations including the Swap Trees transformation; therefore, for brevity, we omit the results on these datasets.

Figure 4.4 shows the results. In general, substantial accuracy improvements were obtained by adding textual information to entity description, as expected. In com-



(a) Nasa, low error (b) Nasa, moderate error (c) Nasa, dirty error

(d) SwissProt, low error (e) SwissProt, moderate error (f) SwissProt, dirty error

(g) PSD, low error (h) PSD, moderate error (i) PSD, dirty error

(j) DBLP, low error (k) DBLP, moderate error (l) DBLP, dirty error

Figure 4.4: Accuracy results on XML data using textual and structural similarity

parison to the experiments reported in Chapter 3, we are now able to get maximum effectiveness in several settings. For example, on Nasa (Figures 4.4(a)–(c)), nearly perfect accuracy is achieved by most similarity functions on Pt2 datasets, even for 50% of error extent. This means that duplicates are properly separated from non-duplicates and positioned on the top of the ranked list (recall the interpretation of the MAP measure in Section 2.4.2). Moreover, effectiveness on PSD (Figures 4.4(g)–(i)) and DBLP

(Figures 4.4(j)–(l)) dramatically increases when textual similarity is employed: compare the MAP values of S, which considers only structure, with those of the other measures.

Further, we observe that the results are stable on PSD and DBLP in the sense that MAP values (except those from S) do not vary too much on a dataset and no similarity function experienced drastic drop in accuracy along different datasets; similar behavior is also observed on Nasa. On the other hand, results on SwissProt (Figures 4.4(d)–(f)) are more unpredictable. The explanation for this behavior lies on the flip side of structural heterogeneity, in particular, label and path diversity: while providing good identifying information for EM tasks, structural heterogenity severely complicates the selection of textual information, and, thus, EDQ query results are more likely to contain spurious PCIs. This fact is evident from the poor accuracy of T, which considers only text, on moderate error (Figure 4.4(e), Pt1) and dirty error (Figure 4.4(f), Pt2) datasets. A closer examination on the path clusters associated to the PCIs returned by the EDS query confirmed that these clusters contain, in fact, several unrelated paths.

All combination approaches were successful in most settings, i.e., they show better accuracy than T and S in isolation. The exceptions occur the on SwissProt dataset — moderate and dirty error extent — owing to the spurious PCIs used for textual representation. For example, in Figure Figure 4.4(e), SLC was unable to leverage the results of S for Pt1, which performed much better than T; likewise, in Figure 4.4(f), TLC exhibits worse results than S for Pt2. Conversely, when S delivered low MAP values such as on PSD and DBLP, all similarity functions performed at least as good as T. This result is similar to the findings of [OC03], whose authors observed that good document representations tend to be robust when combined with other poorly performing representations. Further, the quality of the PC_t set also dictates the accuracy effect of increasing textual information in the tree representation. The results show that the accuracy of similarity functions using textual information increases from Pt1 to Pt2 when the PCI returned by the second path specification is related to a “clean” path cluster — the opposite holds true.

TLC has overall the best accuracy among all similarity functions. Exceptions are noticeable in Figures 4.4(e) and 4.4(f), where TLC is outperformed by SLC on Pt2, and in Figure 4.4(c), where V2 exhibits the best accuracy on Pt1. Nevertheless, TLC has the advantage of being parameter-free, as opposed to SLC which requires the definition of similarity weights, and, in comparison to V2, operating on much shorter profiles. Finally, V2 has the best accuracy among the *epq*-gram versions, more prominently on dirty datasets (e.g, Figure 4.4(l)).

4.6 Related Work

Weis and Naumann [WN05] presented an EM framework in the context of XML datasets. Textual entity descriptions — either pre-defined or heuristically identified — are selected from XML data and stored in relational tables. Then, a sequence of operations is performed towards the identification of duplicates, which basically follow the NFS framework presented in Chapter 2. This approach assumes that all XML trees comply to the same schema; thus, its usage in ad-hoc or "schema-later" settings is impractical.

Milano et al. [MSC06] proposed an edit-distance-based similarity function for unordered trees exploiting textual and structural information. Paths are used to guide textual similarity comparison: two text nodes can be compared only if they have the same path from the root. The set of one-to-one text node matchings between two trees is referred to as *overlay*. The cost of an overlay is defined in terms of the similarity between text nodes and the overall similarity result is obtained from the cost of an optimal overlay — an overlay is optimal if it is not a proper subset of any other overlay and its cost is minimal. The authors presented an algorithm that, first, performs all pairwise similarity calculations and, then, employs a variant of the well-known Hungarian Algorithm to find an optimal overlay. Unfortunately, the algorithm is computationally very expensive — complexity of $O(n^2 \times deg^3)$, for trees with size n and fixed degree deg, which prevents its use in large-scale similarity joins.

Many approaches combining text and structure have been proposed for clustering of XML documents [DG09]. In [TNB08], the structural and textual parts of XML trees are represented by vectors of path and term frequencies, respectively. For the structure, the Euclidean distance is used as similarity function. For the text, the similarity is defined in terms of the semantic association between two trees, which is measured using Singular Value Decomposition [BDF$^+$97]. Similarity results are combined using a linear combination of similarity scores (equivalent to the SLC approach).

There is an extensive research literature on combination of evidence [TSK06, Bis06, Cro00, BM03a]. We have already discussed several combination strategies in Section 2.1.2. In the context of clustering Web pages, Ramage et al. [RHMGM09] exploited user-generated tags from social bookmarking Web sites to improve effectivity. The authors examined several ways of incorporating tags into the bag-of-words representation of Web documents. Tags can be interpreted as structural information. In this regard, the model "Tags as New Words", which treats tags as additional words, bears strong resemblance with TLC. For XML data, we are not aware of any previous work that generates and combines textual and structural tokens in a similar vein as *epq*-grams and PCI_{tlc}.

4.7 Summary

In this chapter, we explored ways of combining evidence from textual and structural XML representations and proposed methods for selecting the textual description of tree-structured entities on heterogenous XML datasets. We proposed tokenization functions that jointly capture textual and structural information for ordered and unordered trees; in this context, we explored approaches to combination of evidence at the token and score levels. The delimitation of text and structure in the representation of XML trees is defined by the user at runtime using the so-called EDS queries.

To support EDS query evaluation, we devised a cluster prototype for each path cluster, i.e., a structure subsuming all paths contained in a cluster. The benefit of cluster prototypes is two-fold: users can specify which part of the documents will constitute their textual representation with vague knowledge about the underlying structure and existing clusters can be updated, as operations such as *insertion of a new document* add a new path to the structural summary. Cluster prototypes are represented as short memory-resident inverted lists. We validate our solutions experimentally and demonstrate that effectiveness can be substantially improved by combining textual and structural similarity.

Chapter 5

Set Similarity Joins

In the previous two chapters, we investigated methods to support EM applications to identify trees representing the same real-world entity but exhibiting structural and textual variations. We have seen that token-based approaches to measuring XML similarity are effective to capture these variations and can outperform competing approaches based on different notions of similarity (e.g., edit distance). In this regard, we have been concerned with the quality and robustness of the token-based similarity functions under study where the method for splitting XML trees into profiles was the most important factor. We now focus on efficiency aspects.

In a similarity join setting, assuming that the (weighted) profiles have already been generated[1], the last step towards the similarity results is the comparison of profiles using a set-overlap-based similarity function. Although such measures are inexpensive to calculate, when dealing with massive datasets, the quadratic cost of comparing every pair of profiles would be exorbitant. Thus, our goal in this chapter is to devise techniques for performing similarity joins efficiently over very large collections of token sets thereby providing scalability.

Tokenization and weighting methods represent an abstraction level to our XML similarity join framework. Instead of node labels, content, relationships, and tree-related operations, the objects that are visible at the next higher layer are sets of tokens, the core operation is set overlap calculation, and the properties of interest for our purpose are set sizes and token frequency distributions. We shall change our terminology to reflect this set abstraction accordingly. We cast the similarity join problem into the *set similarity join* problem and we refer to the corresponding similarity function as a set-overlap-based similarity function.

Besides the large scale, several other issues make the set similarity join problem challenging. The common approach of representing objects as points in a vector space and then exploiting multidimensional data structures is problematic. Vectors as representations of token sets are sparse and have several thousands of dimensions. It is well-known that indexing techniques based on data-space partitioning are ineffective at high dimensions—already for as few as 10 dimensions [WSB98]. Dimension

[1]We discuss the pipelined generation of profiles and set similarity join processing in Chapter 6.

reduction techniques can be employed to make the data more amenable to efficient indexing [BDF+97, YP97]. However, for application domains involving text data, an effective dimensionality in the order of hundreds has to be anticipated even after aggressive dimension reduction (about 90%). For example, the token space dimension of the PCI-based textual profile ($pcl[t](T)$) presented in Chapter 4 is $O(\mathcal{A}^q \times |PC|)$, where $\mathcal{A}$ is the alphabet from which strings are built, q is the q-gram size, and PC is the set of path clusters.[2]

State-of-the-art similarity join algorithms dynamically build an index based on inverted lists for mapping tokens to the sets containing these tokens [SK04, BMS07, XWLY08]. In this context, most algorithms are composed of two main phases: *candidate generation*, which produces a set of candidate pairs, and *verification*, which applies the actual similarity measure to the generated candidates and returns the correct answer. Recently, Xiao et. al [XWLY08] improved the previous algorithm proposed by Bayardo et al. [BMS07] by pushing overlap constraint checking into the candidate generation phase. To reduce the number of candidates even more, the authors proposed a suffix filtering technique, where a relatively expensive operation is carried out, before qualifying a pair as a candidate. For that purpose, the overlap constraint is converted into an equivalent Hamming distance and subsets are verified in a coordinated way using a divide-and-conquer algorithm. As a result, the number of candidates is substantially reduced, often to the same order of magnitude of the result set size.

In this chapter, we propose a new index-based algorithm for set similarity joins. Our work builds upon the previous work of [BMS07] and [XWLY08], however, we follow an opposite approach to that of [XWLY08]. Our focus is on the decrease of the computational cost of candidate generation instead of the reduction of the number of candidates. For this reason, we introduce the concept of *min-prefix*, a generalization of the *prefix filtering* concept [SK04, CGK06] applied to indexed sets. Min-prefix allows to *dynamically* maintain the length of the inverted lists reduced to a minimum, and, therefore, the candidate generation time is drastically decreased. We address the growth of the workload in the verification phase, a side-effect of our approach, by interrupting as early as possible the computation of candidate pairs that will not meet the overlap constraint. We also improve the overlap score accumulation by avoiding the overhead of dedicated data structures. Furthermore, we consider disk-based and parallel versions of the algorithm. Finally, we present a thorough experimental evaluation using synthetic and real-life datasets; the results show that our algorithm consistently outperforms the so-far known ones for unweighted and weighted sets.

This chapter is self-contained and of general interest. Beyond EM, set similarity joins have been used in the context of many other important application areas including text data support in relational databases [Coh98, GIJ+01], collaborative filtering [SSB05], Web indexing [Bro97, TSP08], social networks [SSB05], and information ex-

[2]Note that this analytical bound corresponds to unannotated profiles. For annotated profiles, the space dimension ought to be larger, because multiple appearances of a token in a tree translates into new tokens.

traction [CCGX08]. Hence, we will consider not only XML data in our experiments, but also datasets derived from string data as well as synthetically generated data. In this regard, in addition to the comparison of our algorithm with competing approaches, we also investigate the most important characteristics of the input data affecting the performance of set similarity join algorithms in general. For ease of exposition, we assume unweighted sets in most of the following discussion.

The remainder of this chapter is organized as follows. Section 5.1 defines our terminology and reviews important optimization techniques for set similarity joins. In Section 5.2, we introduce the min-prefix concept and show how it can be exploited to improve runtime of set similarity joins. In Section 5.3, we present further optimizations in the candidate generation and verification phases. Section 5.4 considers disk-based and parallel versions of *mpjoin*, whereas Section 5.5 describes the version for weighted sets. The evaluation of set similarity joins using the PCL_{slc} similarity function, which spans two profiles, is addressed in Section 5.6. Experimental results are presented in Section 5.7. We discuss related work in Section 5.8, before we wrap up with conclusions in Section 5.9.

5.1 Preliminaries

In this section, we first provide background material on set similarity join concepts and techniques. Then, we describe the baseline algorithm for set similarity joins that we use in this work.

5.1.1 Background

Given a finite universe U of tokens and a set collection C, where every set consists of a number tokens from U, let $sf(x_1, x_2)$ be a set similarity function that maps a pair of sets x_1 and x_2 to a number in $[0, 1]$. We assume the similarity function is commutative, i.e., $sf(x_1, x_2) = sf(x_2, x_1)$. Given a threshold τ, $0 \leq \tau \leq 1$, our goal is to identify all pairs $(x_1, x_2), x_1, x_2 \in C$, which satisfy the similarity predicate $sf(x_1, x_2) \geq \tau$.

We focus on a general class of set similarity functions, for which the similarity predicate can be equivalently represented as a set overlap constraint. Specifically, we express the original similarity predicate in terms of an *overlap lower bound* (*overlap bound*, for short) [CGK06].

Definition 5.1 (Overlap Bound)**.** *Let x_1 and x_2 be sets of tokens, sf be a set similarity function, and τ be a similarity threshold. The overlap bound between x_1 and x_2 relative to sf, denoted by $minoverlap(x_1, x_2)$, is a function that maps τ and the sizes of x_1 and x_2 to a real value, s.t. $sf(x_1, x_2) \geq \tau$ iff $|x_1 \cap x_2| \geq minoverlap(x_1, x_2)$.*

Hence, the similarity join problem is reduced to a set overlap problem, where all pairs whose overlap is not less than $minoverlap(x_1, x_2)$ are returned. Table 5.1 shows the overlap bound of the following widely used set similarity functions [SK04,

Table 5.1: Set similarity functions

Function	Definition	$minoverlap(x_1, x_2)$	$[minsize(x), maxsize(x)]$
Jaccard	$\frac{\lvert x_1 \cap x_2 \rvert}{\lvert x_1 \cup x_2 \rvert}$	$\frac{\tau}{1+\tau}(\lvert x_1\rvert + \lvert x_2 \rvert)$	$\left[\tau \lvert x \rvert, \frac{\lvert x \rvert}{\tau}\right]$
Dice	$\frac{2\lvert x_1 \cap x_2 \rvert}{\lvert x_1\rvert + \lvert x_2 \rvert}$	$\frac{\tau(\lvert x_1\rvert + \lvert x_2 \rvert)}{2}$	$\left[\frac{\tau \lvert x \rvert}{2-\tau}, \frac{(2-\tau)\lvert x \rvert}{\tau}\right]$
Cosine	$\frac{\lvert x_1 \cap x_2 \rvert}{\sqrt{\lvert x_1\rvert \lvert x_2 \rvert}}$	$\tau\sqrt{\lvert x_1\rvert \lvert x_2 \rvert}$	$\left[\tau^2 \lvert x \rvert, \frac{\lvert x \rvert}{\tau^2}\right]$

AGK06, LLL08, XWLY08, XWLS09]: Jaccard, Dice, and Cosine. An important observation is that, for all similarity functions, $minoverlap(x_1, x_2)$ increases monotonically with one or both set sizes.

The set overlap formulation enables the derivation of *size bounds*. Intuitively, observe that $|x_1 \cap x_2| \leq |x_1|$ for $|x_2| \geq |x_1|$, i.e., set overlap and, therefore, similarity are trivially bounded by $|x_1|$. By carefully exploiting the similarity function definition, it is possible to derive tighter bounds allowing immediate pruning of candidate pairs whose sizes differ enough.

Definition 5.2 (Set Size Bounds)**.** *Let x_1 be a set of tokens, sf be a set similarity function, and τ be a similarity threshold. The size bounds of x_1 relative to sf are functions, denoted by $minsize(x_1)$ and $maxsize(x_1)$, that map τ and the size of x_1 to a real value, s.t. $\forall x_2$, if $sf(x_1, x_2) \geq \tau$, then $minsize(x_1) \leq |x_2| \leq maxsize(x_1)$.*

Therefore, given a set x, we can safely ignore all sets whose size do not fall within the interval $[minsize(x), maxsize(x)]$. Table 5.1 shows the set size bounds of the aforementioned similarity functions.

Overlap bound and set size bounds give raise to several other optimizations. We can prune the comparison space by exploiting the *prefix filtering* concept [CGK06]. The idea is to derive a new overlap constraint to be applied on subsets of the operand sets. More specifically, for any two sets x_1 and x_2 under a same total order O, if $|x_1 \cap x_2| \geq \delta$, the subsets consisting of the first $|x_1| - \delta + 1$ elements of x_1 and the first $|x_2| - \delta + 1$ elements of x_2 must share at least one element [SK04, CGK06]. We refer to such subsets as *prefix filtering subsets*, or simply *prefixes*, when the context is clear; further, let $pref(x_1)$ denote the prefix of a set x_1, i.e., $pref(x_1)$ is the subset of x_1 containing the first $|pref(x_1)|$ elements according to the ordering O. It is easy to see that, for $\delta = \lceil minoverlap(x_1, x_2) \rceil$, the set of all pairs (x_1, x_2) sharing a common prefix element is a superset of the correct result. Thus, one can identify matching candidates by examining only a fraction of the original sets.

The exact prefix size is determined by $minoverlap(x_1, x_2)$, which varies according to each matching pair. Given a set x_1, a question is how to determine $|pref(x_1)|$

such that it suffices to identify all matchings of x_1 (no false negatives). Clearly, we have to take the largest prefix in relation to all x_2. Because the prefix size varies inversely with $minoverlap(x_1, x_2)$, $|pref\,(x_1)|$ is largest when $|x_2|$ is smallest (recall that $minoverlap(x_1, x_2)$ increases monotonically with $|x_2|$). The smallest possible size of x_2, such that the overlap constraint can be satisfied, is $minsize(x_1)$.

Definition 5.3 (*Max-prefix*). *Let x_1 be a set of tokens. The max-prefix of x_1, denoted by $maxpref\,(x)$, is its smallest prefix needed for identifying $\forall x_2$ s.t. $|x_1 \cap x_2| \geq minoverlap(x_1, x_2)$. The size of max-prefix is given by:*

$$|maxpref\,(x_1)| = |x_1| - \lceil minsize(x_1) \rceil + 1\ . \tag{5.1}$$

Another optimization consists of sorting C in *increasing* order of the set sizes. By exploiting this ordering, one can ensure that x_1 is only matched against x_2, such that $|x_2| \geq |x_1|$. As a result, the prefix size of x can be reduced: instead of $maxpref\,(x)$, we obtain a shorter prefix by using $minoverlap(x, x)$ to calculate the prefix size [BMS07, XWLY08, XWLS09].

Definition 5.4 (*Mid-prefix*). *Let x_1 be a set of tokens. The mid-prefix of x_1, denoted by $midpref\,(x_1)$, is its smallest prefix needed for identifying $\forall x_2 \geq x_1$ s.t. $|x_1 \cap x_2| \geq minoverlap(x_1, x_2)$. The size of mid-prefix is given by:*

$$|midpref\,(x_1)| = |x_1| - \lceil minoverlap(x_1, x_1) \rceil + 1\ . \tag{5.2}$$

Example 5.1. *Recall the qgram profiles from Example 2.7, i.e., $qgram[3](s_1)$ and $qgram[3](s_2)$. Consider JS as similarity function and $\tau = 0.75$. We have $|q(s_1)| = 12$ and $|q(s_2)| = 10$. Then, we have $\lceil minoverlap(q(s_1), q(s_2)) \rceil = \lceil \frac{\tau}{1+\tau}(12 + 10) \rceil = 10$. For $q(s_1)$, we have $[\lfloor minsize(q(s_1)) \rfloor, \lceil maxsize(q(s_1)) \rceil] = [9, 16]$. Further, we have $|maxpref\,(q(s_1))| = 4$ and $|midpref\,(q(s_1))| = 2$. Assuming, for simplicity, that the tokens of $q(s_1)$ are already sorted according to some order O as depicted in Example 2.7, we have $maxpref\,(q(s_1)) = \{$Kai, ais, ise, ser$\}$ and $midpref\,(q(s_1)) = \{$Kai, ais$\}$.*

Feature ordering can be further exploited to improve performance. Because O imposes an ordering on the elements of a set x, we can use the *positional information* of a common token between two sets to quickly verify whether or not there are enough remaining tokens in both sets to meet a given threshold (see [XWLY08], Lemma 1). Given a set $x = \{t_1, \ldots, t_{|x|}\}$, where the subscripts represent the token position in the set, let $rem(x, i)$ denote the number of tokens following the token t_i in x; thus, $rem(x, i) = |x| - i$. We can also rearrange the sets in C according to a specific order, namely the *token frequency ordering*, O_f, to obtain sets ordered by increasing token frequencies. The idea is to minimize the number of sets agreeing on prefix elements and, in turn, candidate pairs by shifting lower frequency tokens to the prefix positions [CGK06].

Algorithm 5.1: The ppjoin algorithm

```
Input: A set collection $C$, a threshold $\tau$
Output: A set $S$ containing all pairs $(x_p, x_c)$ such that $sf(x_p, x_c) \geq \tau$
1  $I(1), I(2), \ldots, I(|U|) \leftarrow \varnothing, S \leftarrow \varnothing$
2  foreach $x_p \in C$ do
3      $M \leftarrow$ empty map from set id to $(os, i, j)$      // os = overlap score
4      foreach $t_i \in maxpref(x_p)$ do                        // candidate generation phase
5          Remove all $(x_c, j)$ from $I(t_i)$ s.t. $|x_c| < minsize(x_p)$
6          foreach $(x_c, j) \in I(t_i)$ do
7              $M(x_c) \leftarrow (M(x_c).os + 1, i, j)$
8              if $M(x_c).os + min(rem(x_p, i), rem(x_c, j)) < minoverlap(x_p, x_c)$
9                  $M(x_c).os \leftarrow -\infty$    // do not consider $x_c$ anymore
10     $S \leftarrow S \cup Verify(x_p, M, \tau)$      // verification phase
11     foreach $t_i \in midpref(x_p)$ do // index $x_p$
12         $I(t_i) \leftarrow I(t_i) \cup \{(x_p, i)\}$
13 return $S$
```

5.1.2 The ppjoin Algorithm

We are now ready to present a "baseline" algorithm for set similarity joins. Algorithm 5.1 shows *ppjoin* [XWLY08], a state-of-the-art, index-based algorithm that comprises all optimizations previously described. Henceforth, we assume that the set collection C is sorted in increasing order of the set sizes as well as each set is sorted according to the total order O_f.

The top-level loop of *ppjoin* scans the dataset C, where, for each set x_p, a *candidate generation phase* delivers a set of candidates by probing the index with the token elements of $maxpref(x_p)$ (lines 4–9). We call the set x_p, whose tokens are used to probe the index, a *probing set*; any set x_c that appears in the scanned inverted lists is a *candidate set* of x_p. Besides the accumulated overlap score, the hash-based map M also stores the token positional information of x_p and x_c (line 7). In the *verification phase*, the probing set and its candidates are checked against the similarity predicate and those pairs satisfying the predicate are added to the result set (line 10). (We defer details about the *Verify* procedure to Section 5.3.1.) Finally, a *pointer* to set x_p is appended to each inverted list $I(t)$ associated with each token t of $midpref(x_p)$ (lines 11 and 12). Note that the algorithm also indexes the token positional information, which is needed for checking the overlap bound (line 8). Additionally, the algorithm employs the lower bound of the set size to dynamically remove sets from inverted lists (line 5).

Note that Algorithm 5.1 has a different notation from that of Xiao et al. [XWLY08] and also present some minor modifications. First, we use the notation introduced in the previous section for overlap bound, size bounds, and prefixes. In the original paper, the authors presented an instantiation of *ppjoin* for Jaccard; for example, given a set x, they used $\tau|x|$ for the lower bound of the set size, whereas, here, we generally use $minsize(x)$. Further, we store positional information in the hash-based map. As we will see shortly, this information is used in the *Verify* procedure to find the position of the last token matched in the candidate generation phase (for both sets of each candidate pair). In their paper, Xiao et al. used the accumulated overlap score to (approximately) obtain this information (see Algorithm 2 in [XWLY08], lines 8 and 12). Finally, we incorporated the mid-prefix optimization in our algorithm. Note that mid-prefix corresponds to the optimization presented in [XWLY08], Lemma 3 (again, instantiated for Jaccard).

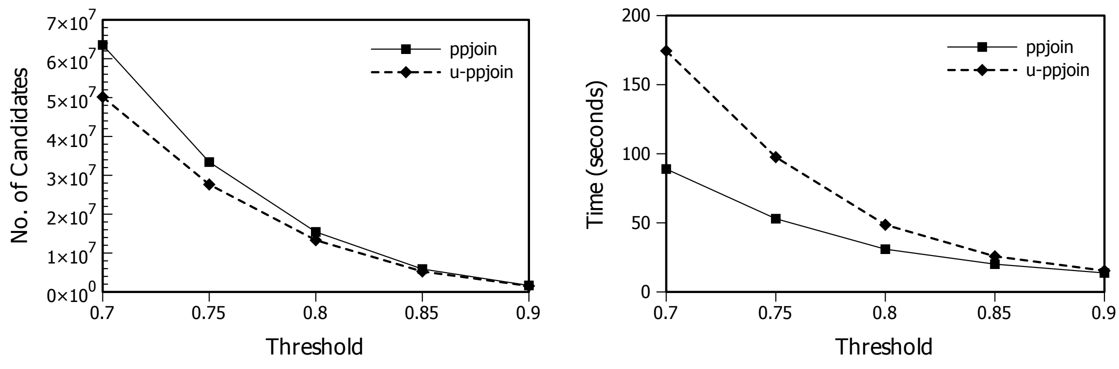
The *ppjoin* algorithm presented above is actually a self-join. Its extension to binary joins is straightforward. We only have to intersect the two sorted inputs as we proceed with the algorithm. Sets of both join partners go through candidate generation, verification, and indexing, as in the self-join case, and we only need to ensure that no set from the same input as the probing set is selected as a candidate. After having finished a join operand, say C_1, we only need to process the other operand (C_2) until we find a set whose size is greater than $maxsize(x)$, where x is the last set of C_1. Note that, after C_1 is completely processed, we do not need to index the elements of C_2. The procedure is quite similar to the disk-based version of set similarity join algorithms, which is discussed in Section 5.4.1. Henceforth, we assume self-joins; binary joins are revisited in Chapter 6.

5.2 Generalizing Prefix Filtering

In this section, we first empirically show that the number of generated candidates can be highly misleading as a measure of runtime efficiency. Motivated by this observation, we introduce the min-prefix concept and propose a new algorithm that focuses on minimizing the computational cost of candidate generation.

5.2.1 Candidate Reduction vs. Runtime Efficiency

Most set similarity join algorithms operate on shorter set representations in the candidate generation phase (e.g., prefixes) followed by a potentially more expensive stage where a thorough verification is conducted on each candidate. Accordingly, previous work has primarily focused on candidates reduction where increased effort is dedicated to candidate generation to achieve stronger filtering effectiveness. In this vein, an intuitive approach consists of moving part of the verification into candidate generation. For example, we can generalize the prefix filtering concept to subsets of any size: $(|x| - \delta + c)$-sized prefixes must share at least c tokens. This idea has already been used for related similarity operations, but in different algorithmic frameworks [LLL08, CCGX08]. Let us examine this approach applied to *ppjoin*. We can easily swap part of the workload between verification and candidate generation by increasing token indexing from $midpref(x)$ to $maxpref(x)$ (Algorithm 5.1, line 11). We call this version *u-ppjoin*, because it corresponds to a variant of *ppjoin* for unordered



(a) No. of candidates: Jaccard on DBLP (b) Runtime efficiency: Jaccard on DBLP

Figure 5.1: Number of candidates vs. runtime efficiency

datasets. Although *u-ppjoin* considers more sets for candidate generation, a larger number of candidate sets are pruned by the overlap constraint (Algorithm 5.1, lines 8 and 9). Figure 5.1(a) shows the results of both algorithms w.r.t. the number of candidates and runtime for varying Jaccard thresholds on a 100K sample taken from the DBLP dataset (details about the datasets are given in Section 5.7). As we see in Figure 5.1, *u-ppjoin* indeed reduces the amount of candidates, especially for lower similarity thresholds, thereby reducing the verification workload[3]. However, the runtime results showed in Figure 5.1(b) are reversed: *u-ppjoin* is considerably slower than *ppjoin*. Similar results were reported by Bayardo et al. [BMS07] for the unordered version of their *All-pairs* algorithm. We also observed identical trends on several other real world datasets as well as for different growth pattern of token indexing. These results reveal that, at least for inverted-list-based algorithms, candidate set reduction alone is a poor measure of the overall efficiency. Moreover, they suggest that the trade-off of workload shift between candidate generation and verification can be exploited in the opposite way.

5.2.2 The Min-prefix Concept

A set x_c is indexed by appending a pointer to the inverted lists associated with tokens $t_j \in midpref(x_c)$, which results in an indexed set, denoted by $I(x_c)$; accordingly, let $I(x_c, t_j)$ denote a token $t_j \in x_c$ whose associated list has a pointer to x_c. A list holds a reference to x_c until being accessed by a probing set x_p s.t. $minsize(x_p) > |x_c|$, when this reference is eventually removed due to size bound checking (Algorithm 5.1, line 5). We call the interval between the indexing of the set x_c and the last set x_p

[3]Actually, the verification workload is even more reduced than suggested by number of candidates. Due to the increased overlap score accumulation in the candidate generation, many more candidates are discarded at the very beginning of the verification phase.

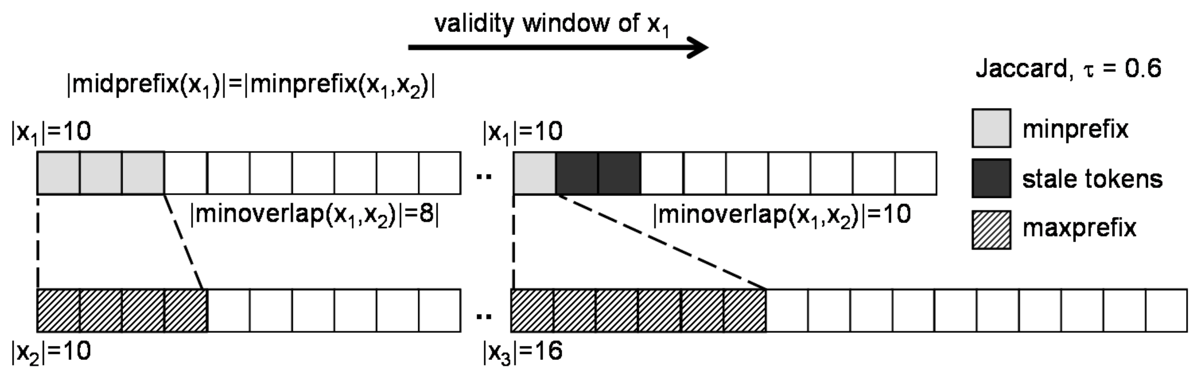


Figure 5.2: Min-prefix example

with $minsize(x_p)$ not greater than x_c the *validity window* of $I\left(x_c\right)$. Within its validity window, any appearance of $I\left(x_c\right)$ in lists accessed by a probing set either elects $I\left(x_c\right)$ as a new candidate, if the first appearance thereof, or accumulates its overlap score.

As previously mentioned, the exact (and minimal) size of $pref\left(x_c\right)$ is determined by the lower bound of pairwise overlap between x_c and a reference set x_p. As our key observation, the minimal size of $pref\left(x_c\right)$ monotonically decreases along the validity window of $I\left(x_c\right)$ due to dataset pre-sorting (size of x_p increases monotonically). Hence, as the validity window of x_c is processed, an increasing number of the indexed tokens in $midpref\left(x_c\right)$ no longer suffices alone to elect x_c as a candidate. More specifically, we introduce the concept of *min-prefix*, formally stated as follows.

Definition 5.5. *(Min-prefix) Let x_c be a set and let $pref\left(x_c\right)$ be a prefix of x_c. Let x_p be a reference set. Then $pref\left(x_c\right)$ is a min-prefix of x_c relative to x_p, denoted as $minpref\left(x_c, x_p\right)$, iff $1 + rem\left(x_c, j\right) \geq minoverlap(x_p, x_c)$ holds for all $t_j \in pref\left(x_c\right)$.*

When processing a probing set x_p, the following fact is obvious: if x_c first appears in an inverted list associated with a token $t_j \notin minpref\left(x_c, x_p\right)$, then $\left(x_c, x_p\right)$ cannot meet the overlap bound. We call a token $I\left(x_c, t_j\right)$, which is not an element of $minpref\left(x_c, x_p\right)$, a *stale token* relative to x_p.

Example 5.2. *Figure 5.2 shows an example with an indexed set $I\left(x_1\right)$ of size 10 and two probing sets x_2 and x_3 of size 10 and 16, respectively. Given Jaccard as similarity function and a threshold of 0.6, we have $midpref\left(x_1\right) = 3$, which corresponds to the number of indexed tokens of $I\left(x_1\right)$. For x_2, we have $minpref\left(x_1, x_2\right) = 3$; thus, no stale tokens are present. On the other hand, for x_3 as reference set, we have $minpref\left(x_1, x_3\right) = 1$. Hence, $I\left(x_1, t_2\right)$ and $I\left(x_1, t_3\right)$ are stale tokens.*

The relationship between the prefix types is shown in Figure 5.3. The three prefixes are minimal in different stages of an index-based set similarity join by exploiting different kinds of information. In the candidate generation phase, the size lower bound

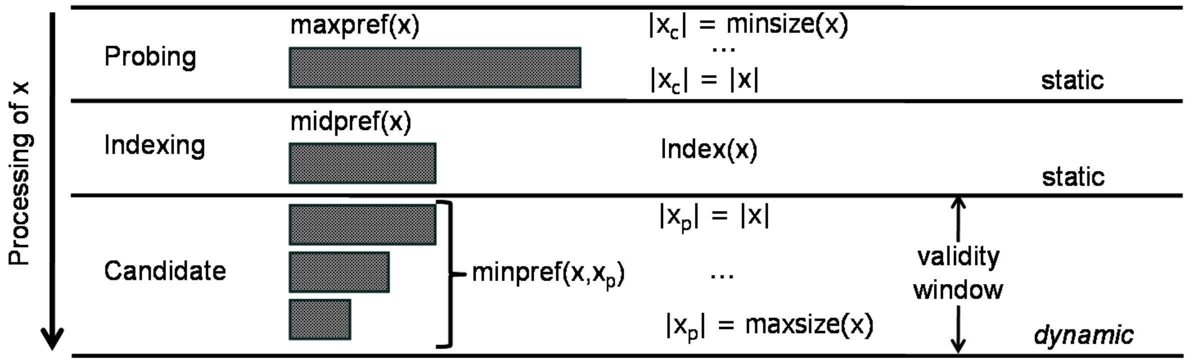


Figure 5.3: Min-prefix generalization of prefix filtering

of a probing set x defines $maxpref(x)$, which is used to find candidates among the (shorter) sets already indexed. To index x, the set collection sort order allows reducing the prefix to $midpref(x)$. The prefixes maxprefix and midprefix are statically defined and fixed-sized. Finally, min-prefix determines the minimum amount of information that needs to *remain* indexed to identify x as a candidate. Differently from the previous prefixes, $minpref(x, x_p)$ is defined in terms of a reference set x_p, which corresponds to the current probing set within the validity window of x; min-prefix is dynamically defined and variable-sized. The following lemma states important properties of stale tokens according to the set collection and the token ordering.

Lemma 5.1. *Let $I(x_c)$ be an indexed set and x_p be a probing set. If a token $I(x_c, t_j)$ is stale in relation to x_p, then $I(x_c, t_j)$ is stale for any $x_{p'}$ such that $|x_{p'}| \geq |x_p|$. Moreover, if $I(x_c, t_j)$ is stale, then any $I(x_c, t_{j'})$, such that $j' > j$, is also stale.*

5.2.3 The mpjoin Algorithm

Algorithm *ppjoin* only uses stale tokens for score accumulation. Candidate pairs whose first common element is a stale token are pruned by the overlap constraint. Because set references are only removed from lists due to size bound checking, repeated processing of stale tokens are likely to occur very often along the validity window of indexed sets. As strongly suggested by the results reported in Section 5.2.1, such overhead in candidate generation can have a negative impact on the overall runtime efficiency.

Listed in Algorithm 5.2, we now present algorithm *mpjoin* which builds upon the previous algorithms *All-pairs* and *ppjoin*. However, it adopts a novel strategy in the candidate generation phase. The main idea behind *mpjoin* is to exploit the concept of min-prefixes to *dynamically reduce* the lengths of the inverted lists to a minimum. As a result, a larger number of irrelevant candidate sets are never accessed and processing costs for inverted lists are drastically reduced.

Algorithm 5.2: The mpjoin algorithm

```
Input: A set collection $C$, a threshold $\tau$
Output: A set $S$ containing all pairs $(x_p, x_c)$ such that $sim\,(x_p, x_c) \geq \tau$
1  $I(1), I(2), \ldots, I(|U|) \leftarrow \varnothing, S \leftarrow \varnothing$
2  foreach $x_p \in C$ do
3      $M \leftarrow$ empty map from set id to $(os, i, j)$    // os = overlap score
4      foreach $t_i \in maxpref(x_p)$ do                      // candidate generation phase
5          Remove all $(x_c, j)$ from $I(t_i)$ s.t. $|x_c| < minsize(x_p)$
6          foreach $(x_c, j) \in I(t_i)$ do
7              if $x_c.prefsize < j$
8                  Remove $(x_c, j)$ from $I(t_i)$    // $I\,(x_c, j)$ is stale
9                  continue
10             $M\,(x_c) \leftarrow (M(x_c).os + 1, i, j)$
11             if $M(x_c).os + min(rem(x_p, i), rem(x_c, j)) < minoverlap(x_p, x_c)$
12                 $M(x_c).os \leftarrow -\infty$    // do not consider $x_c$ anymore
13                 if $M\,(x_c)\,.os + rem\,(x_c, j) < minoverlap(x_p, x_c)$
14                     Remove $(x_c, j)$ from $I(t_i)$    // $I\,(x_c, j)$ is stale
15         $x_c.prefsize \leftarrow |x_c| - minoverlap(x_p, x_c) + 1$// update prefsize
16     $S \leftarrow S \cup Verify(x_p, M, \tau)$    // verification phase
17     $x_p.prefsize \leftarrow |midpref(x)|$    // set prefix size information
18     foreach $t_i \in midpref(x_p)$ do // index $x_p$
19         $I(t_i) \leftarrow I(t_i) \cup \{(x_p, i)\}$
20 return $S$
```

To employ min-prefixes in an index-based similarity join, we need to keep track of the min-prefix size of each indexed set in relation to the current probing set. For this reason, we define min-prefix size information as an attribute of indexed sets, which is named as *prefsize* in the algorithm. At indexing time, *prefsize* is initialized with the size of *midprefix* (line 17). Further, whenever a particular inverted list is scanned during candidate generation, *prefsize* of all related indexed sets is updated using the overlap bound relative to the current probing set (line 15). Stale tokens can be easily identified by verifying if the *prefsize* attribute is smaller than the token positional information in a given indexed set. This verification is done for each set as soon as it is encountered in a list; set references in lists associated with stale tokens are promptly removed and the algorithm moves to the next list element (lines 07–09). Additionally, for a given indexed set, stale tokens may be probed before its *prefsize* is updated. Because tokens of an indexed set are accessed as per the token order by a probing set (they can be accessed in any order by different probing sets though), stale token can only appear as a first common element. In this case, it follows from Definition 5.5 that the overlap constraint cannot be met and the set reference can be removed from the list (lines 13 and 14).

The correctness of *mpjoin* partially follows from Lemma 5.1: it can be trivially

Algorithm 5.3: The *Verify* algorithm

```
   Input: A probing set $x_p$; a map of candidate sets $M$; a threshold $\tau$
   Output: A set $S$ containing all pairs $(x_p, x_c)$ such that $sim(x_p, x_c) \geq \tau$
1  $S \leftarrow \varnothing$
2  foreach $x_c \in M s.t. (overlap \leftarrow M(x_c).os) \neq -\infty$ do
3      if $(t_c \leftarrow tokenAt(x_c, x_c.prefsize)) < (t_p \leftarrow tokenAt(x_p, |maxpref(x_p)|))$
4          $t_p \leftarrow tokenAt(x_p, M(x_c).i + 1), t_c{++}$
5      else
6          $t_c \leftarrow tokenAt(x_c, M(x_c).j + 1), t_p{++}$
7      while $t_p \neq end$ and $t_c \neq end$ do // merge-join-based overlap calc.
8          if $t_p = t_c$ then $overlap{++}, t_p{++}, t_c{++}$
9          else
10             if $rem(min(t_p, t_c)) + overlap < minoverlap(x_p, x_c)$ then break
11             $min(t_p, t_c){++}$    // advance cursor of lesser token
12     if $overlap \geq minoverlap(x_p, x_c)$
13         $S \leftarrow S \cup \{(x_p, x_c)\}$
14 return $S$
```

shown that the inverted-list reduction strategy of *mpjoin* does not lead to missing any valid result. Another important property of *mpjoin* is that score accumulation is done exclusively on min-prefix elements. This property ensures the correctness of the *Verify* procedure, which is described in the next section.

5.3 Further Optimizations

In this section, we discuss the verification phase and propose a modification to *mpjoin* concerning the optimization of overlap score accumulation.

5.3.1 Verification Phase

A side-effect of the index-minimization strategy is the growth of candidate sets. Besides that, as overlap score accumulation is performed only on min-prefixes, larger subsets have to be examined to calculate the complete overlap score. Thus, high performance is a crucial demand for the verification phase. In [XWLY08], token positional information is used to leverage prior overlap accumulation during the candidate generation. We can further optimize the overlap calculation by exploiting the token order to design a merge-join-based algorithm and the overlap bound to define break conditions.

In Algorithm 5.3, we show the *Verify* procedure of *mpjoin*, which applies the optimizations mentioned above. (Note that we have switched to a slightly simplified notation.) The algorithm iterates over each candidate set x_c evaluating its overlap with the probing set x_p. First, the starting point for scanning both sets is located (lines

3–6). The approach used here is similar to *ppjoin* (see [XWLY08] for more details). Note for both sets, the algorithm starts scanning from the token following either the last match of candidate generation, i.e., $i + 1$ or $j + 1$, or the last prefix element, i.e., min-prefix for a candidate set or max-prefix for the probing set. No common token between x_p and x_c is missed, because only min-prefix elements were used for score accumulation during candidate generation. Otherwise, we could have a last match on a stale token, i.e., $x_c.prefsize < j$, and miss another stale token at position $j\prime < j$, whose reference to x_c in the associate inverted list had been previously removed.

The merge-join-based overlap takes place thereafter (lines 7–11). Feature matches increment the overlap accordingly; for each mismatch, the break condition is tested, which consists in verifying if there are enough remaining tokens in the set relative to the currently tested token (line 10). Finally, the overlap constraint is checked and the candidate pair is added to the result if there is enough overlap (lines 12 and 13).

5.3.2 Optimizing Overlap Score Accumulation

Reference [BMS07] argues that hash-based score accumulators and sequential list processing provide superior performance compared to the heap-based merging approach of other algorithms (e.g., [SK04]). We now propose a simpler approach by eliminating dedicated data structures and corresponding operations for score accumulation altogether: overlap scores (and matching positional information) can be stored in the indexed set itself as attributes in the same way as the min-prefix size information. Therefore, overlap score can be directly updated as indexed sets are encountered in inverted lists. We just have to maintain an (re-sizable) array to store the candidate sets, which will be passed to the *Verify* procedure. Finally, after verifying each candidate, we clear its overlap score and matching positional information.

5.4 Practical Aspects

In this section, we address two important practical aspects around our min-prefix approach, namely: a disk-based external version of *mpjoin* to work with limited memory and data splitting for parallel execution.

5.4.1 Disk-Based External Version

So far, we have assumed that there is enough available memory to hold the index through the whole join processing. Obviously, this is an unrealistic assumption for very large datasets. For this reason, we have adapted the “out-of-core” version of All-pairs [BMS07], which conceptually resembles a block nested-loop join: the algorithm makes multiple passes over the input set collection, where a block of the input is indexed and matched as in the in-memory version at each pass. To produce all the results, the algorithm has a matching-only phase where it continues executing the candidate generation phase and verification phase after the last set in a block has

Algorithm 5.4: The out-of-core variant of mpjoin

```
Input: A set collection $C$, a threshold $\tau$, a memory budget parameter
Output: A set $S$ containing all pairs $(x_p, x_c)$ such that $sim\,(x_p, x_c) \geq \tau$
1  $I(1), I(2), \ldots, I(|U|) \leftarrow \varnothing, S \leftarrow \varnothing, lastIndexedSet \leftarrow \varnothing, indexing \leftarrow true,$
2  while $(x_p \leftarrow read()) \neq eof$ do
3      if not indexing and $maxsize(lastIndexedSet) < |x_p|$ then
4          $I(1), I(2), \ldots I(|U|) \leftarrow \varnothing, indexing \leftarrow true$
5          $x_p \leftarrow seek(lastIndexedSet)$
6          continue
7      $M \leftarrow Probe(x_p, \tau)$    // candidate generation phase
8      $S \leftarrow S \cup Verify(x_p, M, \tau)$    // verification phase
9      if indexing then
10         $Index(x_p)$
11         if memory budget exceeded then
12             $indexing \leftarrow false$    // enter matching-only phase
13             $lastIndexedSet \leftarrow x_p$
14 return $S$
```

been indexed until the end of the dataset. Here, we can exploit size bounds to devise a simple yet effective optimization. Instead of proceeding with the matching-only phase until the end of the dataset we can terminate the processing of the current block as soon as a set is read whose size is larger than the size upper bound (*maxsize*) of the last set indexed. From this point, we are sure that no other set will be a valid match of any indexed set.

Algorithm 5.4 shows the disk-based external version of *mpjoin*. The algorithm has an extra parameter specifying the memory budget. Every time this budget is exceeded the algorithm enters in the matching only phase and saves the last set indexed (lines 12 and 13). The main refinement of the algorithm is the stop condition for the matching-only phase (line 3), as described above. After reading a probing set that is large enough, the algorithm clears out the index and start a new block following the last set indexed (lines 4–6).

5.4.2 Parallel Execution

The external version of *mpjoin* can process arbitrarily large amount of data. Nevertheless, some sort of parallelism is necessary for dealing with massive datasets. As for the out-of-core version, size bounds provide a natural way to split the input data among multiple processors and memories. This approach was adopted by Theobald et al. [TSP08] in their parallel algorithm; the underlying technique is basically the same as that used by Arasu et al. [AGK06] to divide a Jaccard-based set similarity join instance into a set of smaller Hamming-based ones. In the following, we briefly review this size-based data splitting strategy and discuss its use with *mpjoin*.

First, the set of integers is partitioned into P, where each $P_i \in P$ is defined by the interval $[l_i, r_i]$. Specifically, starting from $P_1 = [1, 1]$, define $P_i = [r_{i-1} + 1, \lfloor maxsize(l_i) \rfloor]$, where $maxsize(l_i)$ is the size upper bound value obtained from a set of size l_i. Next, set collections $C_1, C_2, \ldots$ of C are constructed as follows: for each set $x \in C$, if $|x| \in P$, then add x to C_i and C_{i+1}. It can be shown that if $x_1 \in C_i$ and $sf(x_1, x_2) \geq \tau$, then $x_2 \in C_{i-1} \cup C_i \cup C_{i+1}$ (see [AGK06] and [TSP08] for details).

We can execute instances of *mpjoin* on each collection C_i independently. Only a minor modification on *mpjoin* is needed to avoid duplicate result pairs. Given a collection C_i, the algorithm starts with an indexing-only phase, where incoming sets are directly indexed (Algorithm 5.2, lines 17–19) without executing the candidate generation phase and verification phase—contrast this stage with the matching-only phase of the out-of-core variant. The indexing-only phase continues until the first set is seen whose size is no shorter than l_i; afterwards, the algorithm switches to its normal operation. The reason for the indexing-only phase is that the sets shorter than l_i are processed by the mpjoin instance associated with C_{i-1}.

As a consequence of the data splitting, each instance of *mpjoin* processes input data exhibiting more concentrated set size distribution. As we will empirically demonstrate in Section 5.7, this fact reduces the performance gains of *mpjoin*—and *ppjoin* as well. Note, however, that some subsets C_i may contain very few elements. Hence, in practice, contiguous subsets will be merged to form the input of a *mpjoin* instance. Devising a subset merging strategy that maximizes both performance of set similarity joins and parallelism is a topic for future work.

5.5 The Weighted Case

We now consider the weighted version of the set similarity join problem. In this version, sets are drawn from a universe of tokens U_w, where each token t is associated with a weight $w\,(f)$. As defined in Chapter 2, the weighted size of a set x, denoted as $w\,(x)$, is given by the summation of the weight of its elements, i.e., $w\,(x) = \sum_{t \in x} w\,(t)$. Correspondingly, the weighted Jaccard similarity (WJS), for example, is defined as $WJS\,(x_1, x_2) = w\,(x_1 \cap x_2)/w\,(x_1 \cup x_2)$. All concepts presented in Section 5.1 can be easily modified to accord with weighted sets. In particular, the prefix definition has to be slightly modified. Given an overlap bound δ, the weighted prefix of a set x, denoted as $pref\,(x)$, is the *shortest* subset of x such that $w\,(pref\,(x)) > w\,(x) - \delta$.

We now present the weighted version of *mpjoin*, called *w-mpjoin*. The most relevant modifications are listed in Algorithm 5.5. As main difference to *mpjoin*, *w-mpjoin* uses the sum of all token weights up to a given token instead of token positional information. For this reason, we define the *cumulative weight* of a token $t_i \in x$ as $c\,(t_i) = \sum w\,(t_j)$, where $1 \leq j \leq i$. We then index $c\,(t_i)$, for each $t_i \in midpref\,(x)$ and set *prefsize* to the cumulative weight of the last token in $midpref\,(x)$ (lines 18–22). Note that token positional information is still necessary to find the starting point of scanning in the *Verify* procedure.

The utility of the cumulative weight in the candidate generation is twofold. First, it

Algorithm 5.5: The w-mpjoin algorithm

```
Input: A weighted set collection $C$, a threshold $\tau$
Output: A set $S$ containing all pairs $(x_p, x_c)$ such that $sim(x_p, x_c) \geq \tau$
1  ...
6  foreach $t_i \in maxpref(x_p)$ do                        // candidate generation phase
7      Remove all $(x_c, c(t_j), j)$ from $I(t_i)$ s.t. $w(x_c) < minsize(x_p)$
8      foreach $(x_c, c(t_j), j) \in I(t_i)$ do
9          if $x_c.prefsize + w(t_j) < c(t_j)$
10             Remove $(x_c, c(t_j), j)$ from $I(t_i)$      // $I(x_c, c(t_j), j)$ is stale
11             continue
12         $M(x_c) \leftarrow (M(x_c).os + w(t_j), i, j)$
13         if $M(x_c).os + min(crem(x_p, i), crem(x_c, j)) < minoverlap(x_p, x_c)$
14             $M(x_c).os \leftarrow -\infty$     // do not consider $x_i$ anymore
15             if $M(x_c).os + crem(x_c, j) < minoverlap(x_p, x_c)$
16                 Remove $(x_c, c(t_j), j)$ from $I(t_i)$        // $I(x_c, c(t_j), j)$ is stale
17         $x_c.prefsize \leftarrow w(x_c) - minoverlap(x_p, x_c)$
18     $S \leftarrow S \cup Verify(x_p, M, \tau)$     // verification phase
19     $cweight \leftarrow 0$
20     foreach $t_i \in midpref(x_p)$ do // index $x_p$
21         $cweight \leftarrow cweight + w(t_i)$
22         $I(t_i) \leftarrow I(t_i) \cup \{(x_p, cweight, i)\}$
23     $x_p.prefsize \leftarrow cweight$
24 ...
```

is used for overlap bound checking. Given $c(t_i)$, the cumulative weight of the tokens *following* t_i in x is $crem(x, i) = w(x) - c(t_i)$. Hence, $crem$ can be used to verify whether or not there are enough remaining cumulative weight to reach the overlap bound (lines 12 and 14). Second, the cumulative weight is used to identify stale tokens by comparing it with *prefsize* (line 08). Note that the cumulative weight of the last token in $minpref(x_c, x_p)$ is always greater than the current *prefsize*. Hence, to be sure that a given token is stale, we have to add the weight of the current token to *prefsize* before comparing it to the cumulative weight.

For brevity, we do not discuss the weighted version of the Verify procedure, but the modifications needed are straightforward.

5.6 Evaluation Using Multi-Set Representation

Among the similarity functions described in Chapter 4, EPQ and PCL_{tlc} are defined over a single profile and, therefore, can be straightforwardly employed in set similarity joins. On the other hand, PCL_{slc} entails the profiles $pcl[s]$ and $pcl[t]$ and the corresponding weights λ_p and λ_s. To evaluate our algorithms using PCL_{slc} as similarity function, we adopt a *multi-set representation*[4], in which the sets derived from the two profiles are "carried" through the similarity join processing. The algorithm is then normally executed on either the sets derived from $pcl[t]$ or $pcl[s]$. We call these sets *primary sets*. In the verification phase, for those primary sets satisfying the overlap constraint, we calculate the similarity of the other set, which is called *secondary set*. Finally, we calculated the linear combination of the resulting similarity values and test it against the threshold.

The above approach corresponds to evaluating the conjunction $sf_p \geq \tau_p \bigwedge sf_s \geq \tau_s$, where sf_p and τ_p (sf_s and τ_s) are the similarity value and the threshold related to the primary (secondary) set, respectively. The threshold τ_p is used instead of the original threshold τ in the regular similarity join processing on the primary sets; therefore, it has to be adjusted in order to avoid missing valid results. Specifically, the value of τ_p is given by:

$$\tau_p = \frac{\tau - \lambda_s}{\lambda_p}\ , \tag{5.3}$$

where λ_p and λ_s are the weights of sf_p and sf_s. For example, consider $\tau = 0.8$ and $\lambda_p = \lambda_s = 0.5$. The value of τ_p is therefore 0.6. Note that Equation 5.3 restricts the space of combinations of τ, λ_p, and λ_p. For instance, in the previous example, the value of τ must be no less than 0.5. Moreover, because λ_s must be less than τ, the value of the weights may determine which sets must be used as secondary sets (i.e., whether the sets derived from $pcl[t]$ or $pcl[s]$).

Aside from restrictions on the value of thresholds and weights, the choice of the primary set is an important performance factor. In general, the choice must be based on which similarity predicate leads to the most efficient set similarity join evaluation. From Equation 5.3, we can easily see that using the set associated with the higher weight as primary set results in higher τ_p values. It is well-known that set similarity joins executes faster with higher thresholds and, obviously, are more selective. Nevertheless, we will see in the next section that, besides the threshold, characteristics of the dataset also highly influence the performance of the algorithms. Moreover, different from the case where the predicates considered are expensive (recall the metric for expensive-predicate ordering in Equation 2.5 on Page 27), the influence of selectivity is reduced due to the low cost of set similarity computation. In Section 5.7.6, we compare the performance of using sets derived from $pcl[t]$ and $pcl[s]$ as primary predicate. A cost model for selecting the primary set is sketched in Chapter 6.

5.7 Experiments

In this section, we present and discuss the results of our experimental study. The goals of our experiments are:

[4]Multi-set in this context should not be confused with multiset (aka bags) which allows an element to appear more than once.

1. To measure the runtime performance of our algorithms, *mpjoin* and *w-mpjoin*, and compare them against previous, state-of-the-art set similarity join algorithms.

2. To identify the most important characteristics of the input data and input parameters driving the performance of the set similarity joins algorithms under study.

3. To evaluate the scalability of the algorithms and their respective out-of-core variant.[5]

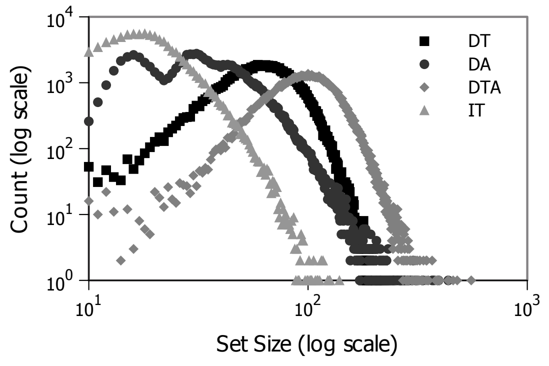
4. To measure and compare the performance achieved by our algorithms on XML data using the similarity functions presented in Chapter 4.

For goals 1–3, we conducted our study under several different data distributions and configuration parameters using real-world datasets derived from string data as well as synthetic datasets. We then focused on XML and for goal 4 we used the same datasets of the accuracy experiments reported in the previous chapters. All tests were run on an Intel Xeon Quad Core 3350 2,66 GHz, about 2.5 GB of main memory, and using Java Sun JDK 1.6.0.

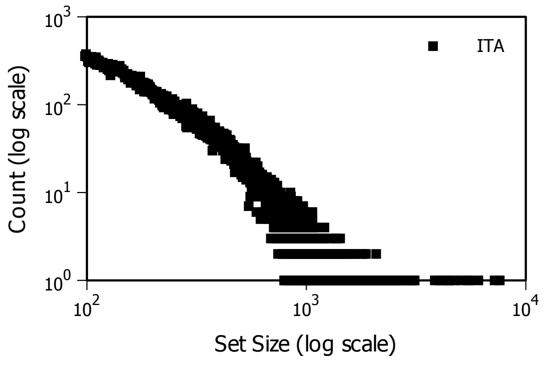
5.7.1 Algorithms

We focused on index-based algorithms, because they consistently outperformed competitor signature-based algorithms [BMS07] (see discussion in Section 5.8) and implemented the best known index-based algorithms due to Xiao et al. [XWLY08]. For unweighted sets, we used *ppjoin+*, an improved version of *ppjoin*, which applies a suffix filtering technique in the candidate generation phase to substantially reduce the number of candidates. This algorithm constitutes an interesting counterpoint to *mpjoin*. We also explored a *hybrid* version, which combines *mpjoin* and *ppjoin+* by adding the suffix filtering procedure in *mpjoin* (Algorithm 5.2, inside the loop of line 6 and after line 14). As recommended by the authors, we performed suffix filtering only once for each candidate pair and limited the recursion level to 2. For weighted sets, however, it is not clear how to adapt the suffix filtering technique, because the underlying algorithm largely employs set partitioning based on subset size. In contrast, when working with weighted sets, cumulative weights have to be used, which requires subset scanning to calculate them also for unseen elements. For this reason, this approach is likely to result in very poor performance. Therefore, we refrained from using *ppjoin+* and instead employed our adaptation of *ppjoin* for weighted sets, denoted *w-ppjoin*. For evaluation of weighted sets, we used the well-known IDF weighting scheme. For brevity, we only report results for the Jaccard similarity. The corresponding results for other similarity functions follow identical trends. In the experiments, we focus on

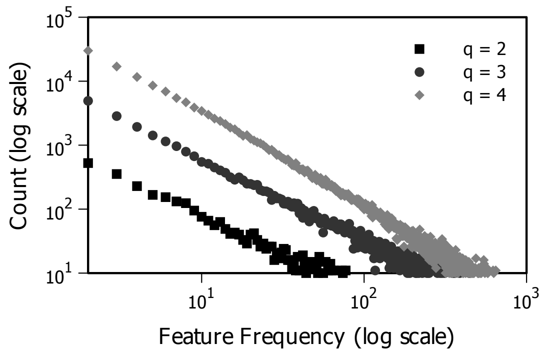
[5]We do not consider the parallel version of the algorithms in our experiments. The relative performance of the algorithms under parallel execution can be nevertheless roughly estimated from the results presented in this section.



(a) Set size distribution, *DBLP* and *IT* datasets



(b) Set size distribution, *ITA* dataset



(c) Count-frequency plot of tokens for q = 2–4, *DTA* dataset

Figure 5.4: Set size and and token frequency distributions

evaluating the performance gains obtained from the optimized candidate generation phase, in particular, the effectiveness of the min-prefix technique. Therefore, we used the improved verification procedure in all algorithms (see Section 5.3.1).

5.7.2 Datasets

Here, we describe the string data and synthetic datasets; XML datasets are described along with its respective experiments in Section 5.7.6. For string data, we used DBLP and *IMDB*[6] containing information about movies. To obtain different data distributions, we first derived three subsets from each dataset by selecting different fields from them: title (*DT*), author (*DA*), and their concatenation (*DTA*), for *DBLP*; title (*IT*), actor (*IA*), and their concatenation (*ITA*), for *IMDB*. We randomly sampled strings from the corresponding fields, converted them to upper-case letters and eliminated

[6]www.imdb.com

Table 5.2: Parameters used in the experiments

Parameter	μ	σ	α	q	N	τ
Description	normal mean	normal sdev	Zipf exponent	q-gram size	no. of input sets	threshold
Range	$[50, 300]$	$[10, 50]$	$[0.6, 2]$	$[2, 4]$	$[0.1M, 1M]$	$[0.5, 0.9]$
Default	100	25	1	2	$0.1M$	0.75

repeated white spaces. Each string is converted into a set of tokens by tokenizing it into sets of q-grams and using the Karp–Rabin fingerprint function [KR87] to map each q-gram to a hash value (with small probability of collision). We then ordered the tokens within a set according to their frequency, and stored the sets in ascending size order. Figures 5.4(a) and 5.4(b) plot the set size distribution. The distribution values of the *DBLP* datasets and *IT* fit reasonably well to a log-normal model (note the log scale on both axes), whereas those of *ITA* and *IA* (only *ITA* is shown) resembles a power-law relationship. Besides the set size, we also vary the token frequency distribution of the datasets by using differing q-gram sizes. Figure 5.4(c) shows the "count-frequency plot"[7] for q ranging from 2 to 4 on the *DTA* dataset—we obtained similar results with the other datasets. The distributions seem to follow a power-law distribution with exponent α about 1.125, 1.194, and 1.509, for $q = 2$, $q = 3$, and $q = 4$, respectively,[8] i.e., the skewness increases with the size of q.

In addition to datasets derived from string data, we generated synthetic set collections to have closer control over data distribution and to support our conclusions on real datasets. The details of data generation process are as follows: we first created N sets and inserted them into a list L. Then, we generated one unique token value v (using sequential numbers) at a time together with its frequency f; for each generated token v, we randomly selected f sets from L and insert a copy of v into each of them. When a set was entirely filled, we removed it from L and we continued the process until L is empty. Set sizes were drawn from a normal distribution and token frequency from a Zipf distribution [GSE+94]. Table 5.2 summarizes the parameters used for data generation as well as those regarding the input of the similarity join algorithms, i.e., number of input sets and threshold. We used the default value unless stated otherwise.

[7]For this plot, we excluded frequencies with count less than 10 to avoid fluctuation effects.

[8]We used a traditional and simple procedure to model the token frequency distribution with a power-law distribution: we fit a straight line on the (log-log) "count-frequency plot" using least-square linear regression and took the absolute slope of the straight line as the exponent α. Note that accurately estimating α as well as making a strong case for a power-law distribution against competing distributions is a difficult problem. In [CSN09], the authors propose maximum likelihood estimators and goodness-of-fit tests based on the Kolmogorov–Smirnov measure and likelihood ratios. Here, an approximate modeling is nevertheless sufficient for the purposes of our discussion.

5.7.3 Performance Results on Synthetic Datasets

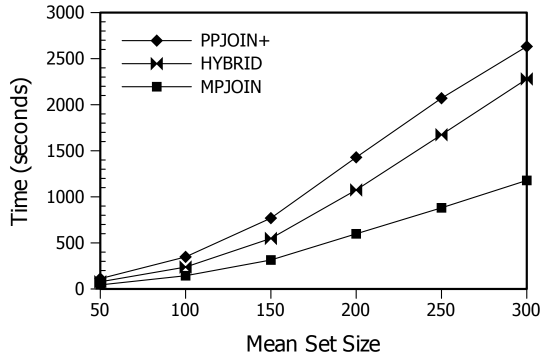
We first analyze the performance of the algorithms under controlled data distribution parameters, namely, mean and standard deviation of the set sizes, and skew of the token frequency distribution. We start with the results for the unweighted version of the algorithms, which are shown in Figures 5.5(a)–(c). In all settings, *mpjoin* clearly exhibits the best performance. In particular, *mpjoin* achieves performance gains compared to *ppjoin+* about a factor of 2.5 on average. Although the *hybrid* version outperforms *ppjoin+*, it is about 70% slower than *mpjoin* on average. Evidently, the candidate reduction does not pay-off the extra-effort of the suffix filtering (we emphasize this observation when we detail the workload of the algorithms).

The performance of all algorithms severely degrades as the mean set size increases (Figure 5.5(a)). This effect is not a surprise, because larger sets translate into larger prefixes and therefore more tokens are used to probe the index in the candidate generation phase. Moreover, larger subsets have to be processed in the verification phase. Another crucial aspect is inverted list reduction. Because we increased the mean of the set size distribution while maintaining its standard deviation constant ($\sigma = 25$), dynamic removal of indexed sets by size bound and min-prefix checking turned out to be less effective. For example, consider a set x with size 300, therefore $minsize(x) = 225$. For $\mu = 300$, nearly all the sets have sizes not smaller than $minsize(x)$, i.e., sizes are at most three standard deviations smaller than the mean size (recall that set sizes are normally distributed in this experiment). Likewise, we have $maxsize(x) = 400$; hence, the validity window of x will last until the end of the dataset. In this connection, it is easy to see that the worst-case scenario is an equi-sized set collection: $minpref(x)$ would be equal to $midpref(x)$ along the whole validity window of x and $minsize(x)$ checking would be useless, i.e., dynamic index reduction would not be possible.

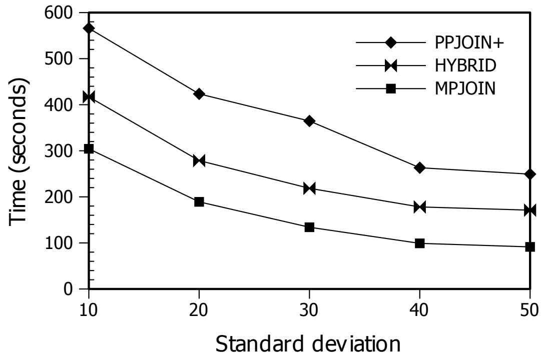
Figure 5.5(b) plots the results with varying σ. All algorithms run significantly faster as the standard deviation increases, which confirms the influence of the size distribution spread on performance. Particularly, the performance gain of *mpjoin* over *ppjoin+* increases from 1.8 to 2.7 as σ increases from 10 to 50. This improvement is due to the increased number of set entries associated with stale tokens in the index that degrades the performance of *ppjoin+*, but are removed by *mpjoin*.

Figure 5.5(c) shows the results with varying skew. Again, *mpjoin* achieves more than twofold speed-ups on average over *ppjoin+*. Furthermore, the runtime of all algorithms is drastically reduced as the skew increases. The reason for this improvement is that there are an increased number of low-frequency tokens with higher skew, which are placed at the prefixes due to token frequency ordering. As a result, the inverted lists are shorter and there is much less prefix overlap between dissimilar sets, thereby decreasing the number of generated candidates.

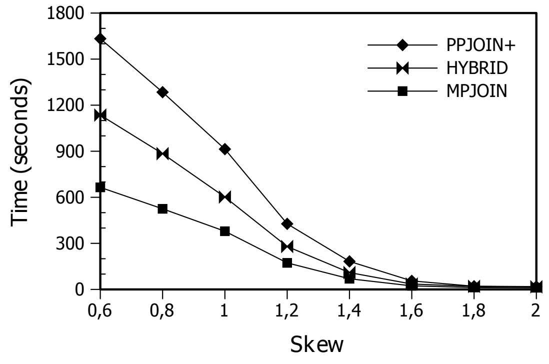
Figures 5.5(d)–(f) plot the results for weighted sets. As for unweighted sets, *w-mpjoin* is faster than *w-ppjoin* in all settings achieving up to twofold speed-ups. Notably, the algorithms are considerably faster than those for the unweighted case, because the weighting scheme results in shorter prefixes. In general, we observe the



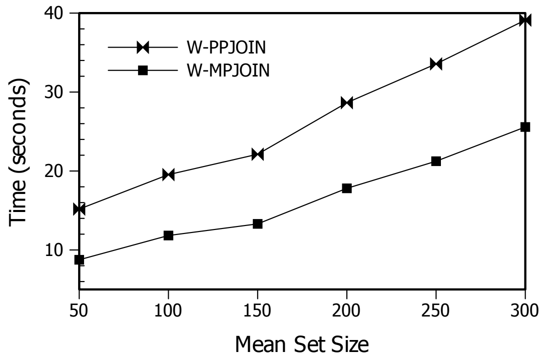
(a) Unwei. sets, varying μ



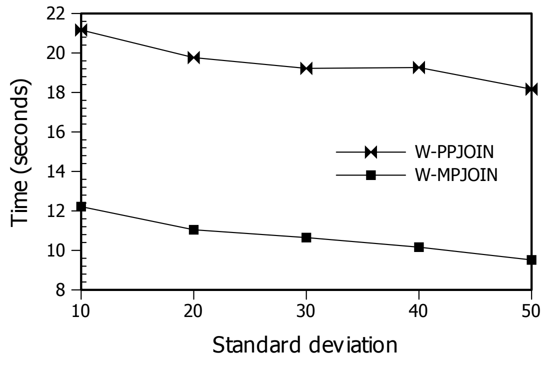
(b) Unwei. sets, varying σ



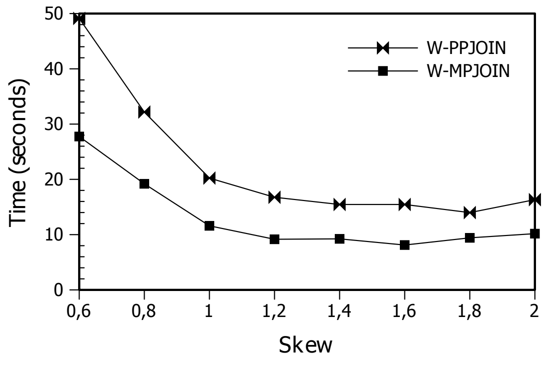
(c) Unwei. sets: varying α



(d) Weighted sets, varying μ



(e) Weighted sets, varying σ



(f) Weighted sets, varying α

Figure 5.5: Performance results on synthetic data

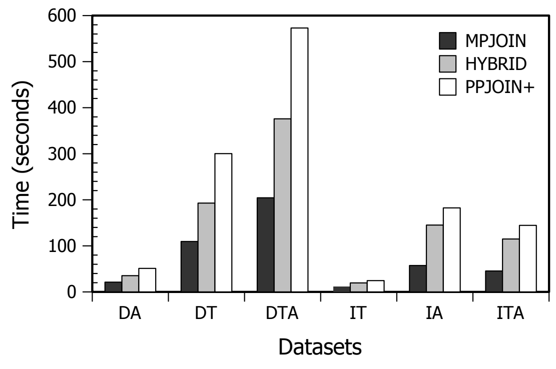
same trends for weighted sets: performance worsens with larger sets, but improves at higher set size variance and token frequency skew. However, data distribution variation affects the algorithms by a lesser degree owing to the reduced prefix size.

5.7.4 Performance Results on Real Datasets

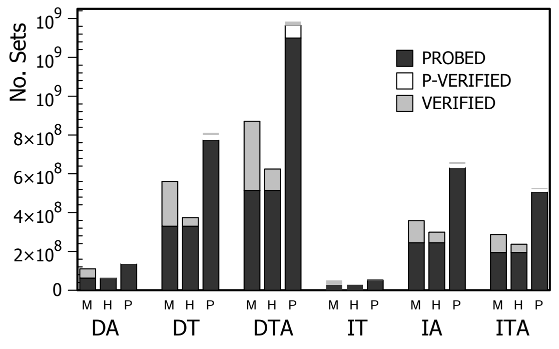
We now analyze the efficiency of the set similarity join algorithms using real datasets. Figure 5.6(a) shows the runtime performance using unweighted sets. The trends observed are similar to those on the synthetic data: *mpjoin* is the best, *ppjoin+* the worst. On all datasets, *mpjoin* is more than two times faster than *ppjoin+*, achieving more than a threefold speed-up over *ppjoin+* on *IA* and *ITA* datasets.

The reasons for the above results are revealed in Figure 5.6(b), which illustrates the workload on the candidate generation and verification phases, i.e., the number of sets processed in these phases. In the chart, *PROBED* corresponds to indexed sets appearing in the inverted lists during the candidate generation phase. Because of the partial overlap score accumulation when generating candidates, some sets are immediately pruned at the first overlap bound checking in the verification phase without further processing (see Algorithm 5.3, line 10); these sets are represented by *P-VERIFIED*. Finally, *VERIFIED* corresponds to the sets that are actually scanned in the verification phase including those that will be part of the output. Note that *mpjoin*, *hybrid*, and *ppjoin+* are abbreviated in the charts by *M*, *H*, and *P*, respectively. In the candidate generation phase, *ppjoin+* doubled the number of indexed sets needed by *mpjoin*. The extra indexed sets of *ppjoin+* are related to stale tokens, i.e., irrelevant candidates that are repeatedly considered along their validity window. Together with the elimination of dedicated data structures for score accumulation, the decreased number of sets processed by *mpjoin* dramatically reduces the computational cost for candidate generation. As expected, the number of sets delivered to the verification phase by *mpjoin* is larger. Moreover, because fewer tokens are considered, score accumulation is reduced when generating candidates. As a result, *P-VERIFIED* is negligible for *mpjoin*, i.e., nearly all sets passed on to the verification phase have to be processed. But now the optimization employed in the verification phase (see Section 5.3.1) comes into play and the higher workload does not translate into an overwhelming performance penalty. For instance, consider the *DTA* data set. Even though *VERIFIED* for *mpjoin* is about 18x larger compared to *ppjoin+*, the overall execution runtime of *mpjoin* is about 2.8x shorter. The advantage of faster candidate generation is made explicit when comparing *mpjoin* to *hybrid*. *PROBED* is the same for both algorithms, but *VERIFIED* is about 5.6x shorter for hybrid due to suffix filtering. However, this saving is ineffective: the overall runtime of *hybrid* is about the double of that of *mpjoin*.

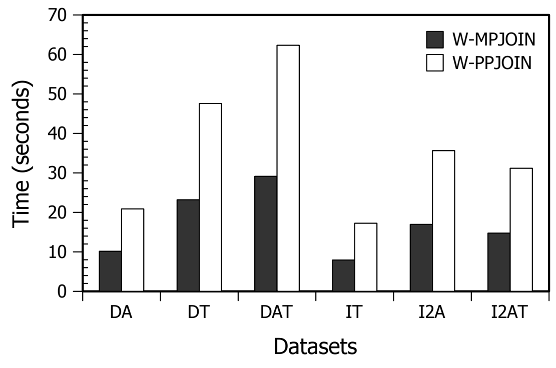
The performance of the algorithms across the datasets is dictated by the underlying data distribution. The set size distribution of the *DBLP* datasets and *IT* have similar shape (see Figure 5.4(a)) and the runtime of the algorithms closely follow the mean set size. On the other hand, all algorithms are faster on *IA* and *ITA* compared to *DT* and *DTA*. Although *IA* and *ITA* contain some very large sets, their set size distribution is more dispersed than those of *DA* and *DTA*. As a result, the validity window of indexed sets is shorter and more entries in the inverted lists are removed due size bound checking. Also, tokens become stale more quickly within the validity window, which favors *mpjoin*: the performance gap between *mpjoin* and *ppjoin+* is larger on *IA* and *ITA*.



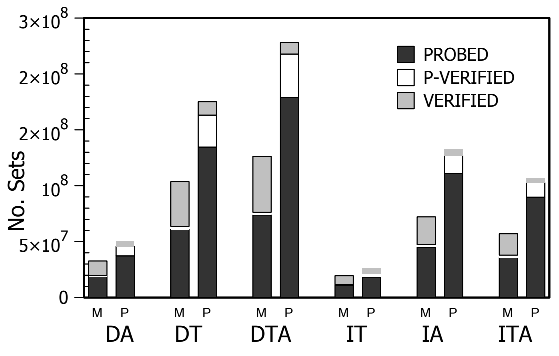
(a) Runtime, unweighted sets



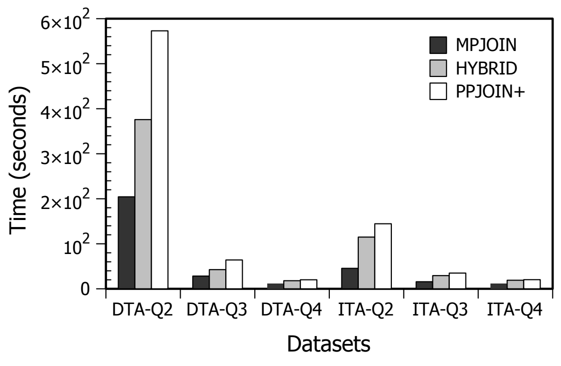
(b) Workload, unweighted sets



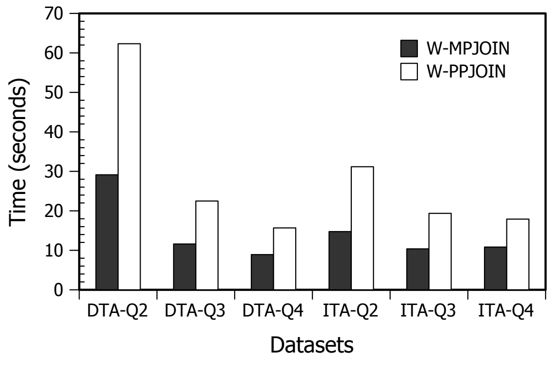
(c) Runtime, weighted sets



(d) Workload, weighted sets



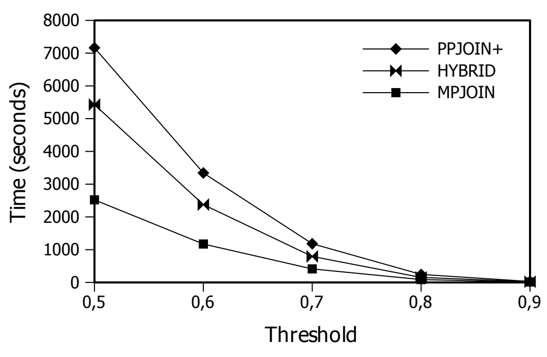
(e) Varying q size, unweighted sets



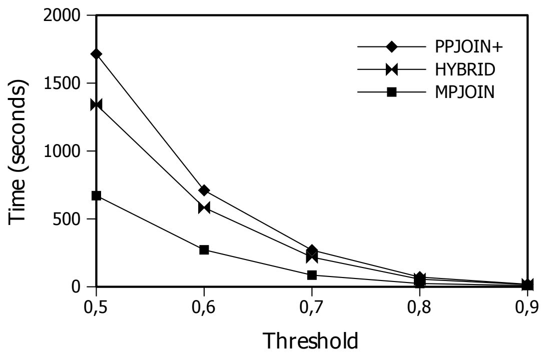
(f) Varying q size, weighted sets

Figure 5.6: Performance results on real data

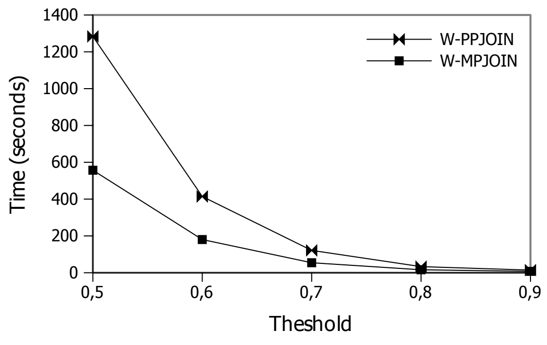
Figure 5.6(c) shows the results for weighted sets. The trends are similar to the ones for unweighted sets. *w-mpjoin* outperforms *w-ppjoin* by a factor larger than 2 in all measurements and the performance of the algorithms across the datasets follows the respective data distribution. As observed on the synthetic datasets, all algorithms are



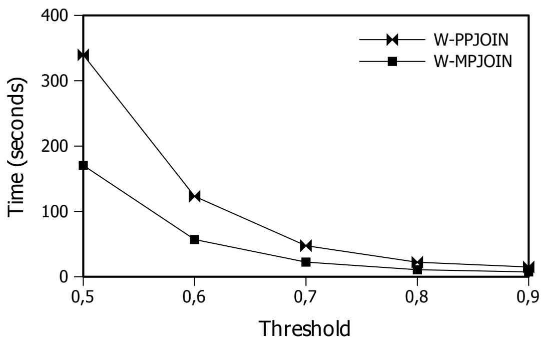
(a) DTA dataset, unweighted sets



(b) ITA dataset, unweighted sets



(c) DTA dataset, weighted sets



(d) ITA dataset, weighted sets

Figure 5.7: Performance results with varying threshold

much faster on weighted sets (about one order of magnitude), which is explained by the lower workload owing to shorter prefixes (see Figure 5.6(d)).

We also analyzed the performance of the algorithms under different token frequency distributions by varying the size of q. As mentioned previously, the skew increases with the size of q. Figure 5.6(e) and Figure 5.6(f) show the results on *DTA* and *ITA* datasets for unweighted and weighted sets, respectively—we obtained similar results on the other datasets. As for synthetic data, all algorithms become much faster as the data becomes more skewed. Note, the performance advantage of *mpjoin* is more prominent when the skew is lower. For instance, on the ITA dataset, the performance gains increase from about 1.7x with $q = 4$ to 3.1x with $q = 2$, for unweighted sets; for weighted sets, the increase is from 1.7x to 2.1x. This observation is of particular importance, because many application domains are characterized by relatively low-skewed data. For example, [CHK$^+$07] recommends q-grams of size 2 to obtain the best quality results in a data cleaning scenario.

Finally, we measured the runtime performance with varying threshold parameter. Figures 5.7(a) and 5.7(b) show the results for unweighted sets on the *DTA* and

ITA, respectively; Figures 5.7(c) and 5.7(d) show the results on the same datasets for weighted sets. *mpjoin* and *w-mpjoin* remain faster than their competitors throughout the whole threshold range on both datasets. All algorithms considerably increased their runtime as the threshold decreases (two orders of magnitude from 0.9 to 0.5), mainly because lower thresholds imply larger prefixes.

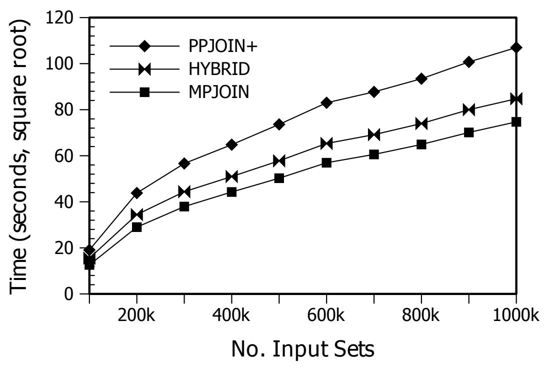
5.7.5 Scalability Experiments

We conducted scalability tests on datasets varying from 100K to 1000K in steps of 100K. We also evaluated the performance of the disk-based version of the algorithms by restricting the memory budget such that only 200K input sets could be dynamically indexed and kept memory-resident; for larger numbers of input sets, the algorithms had to scan the disk-resident input data multiple times to complete the operation as described in Section 5.4.1. We report the results on synthetic data with default parameters (e.g., $\mu = 100$, $\sigma = 25$, $\theta = 1$) and on the *DTA* dataset. The algorithms were configured to stop indexing and enter the matching-only phase after indexing (0.35M) 1.2M tokens for (weighted) unweighted sets on the synthetic dataset; (1.3M) 3.1M tokens for (weighted) unweighted sets on the *DTA* dataset (these values roughly reflect the number of tokens indexed on an input containing 200K sets).

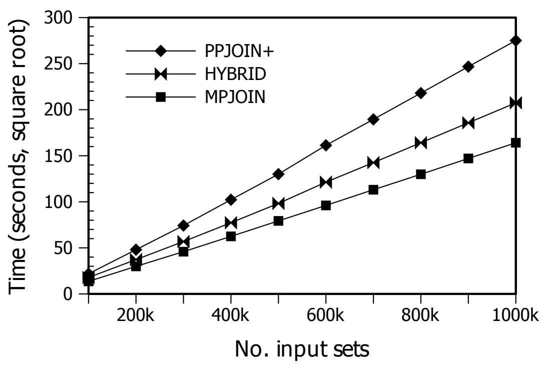
Figures 5.8(a) and 5.8(b) show the results for unweighted sets on the synthetic dataset and *DTA*, respectively; Figures 5.8(c) and 5.8(d) show the results on the same datasets for weighted sets. The runtime of all algorithms on both datasets exhibits a quadratic growth (note that we show the square root of the runtime). This behavior is expected because the workload in terms of set candidates maintained and processed by set similarity join algorithms also grows quadratically with the input size (also observed in [XWLY08]). The growth rate of all algorithms is quite similar and their relative performance practically stays constant as the input size increases. Finally, the IO overhead of the disk-resident variant incurs only a little performance penalty as we do not observe any significant degradation on input data containing more than 200K sets.

5.7.6 Performance Results on XML Data

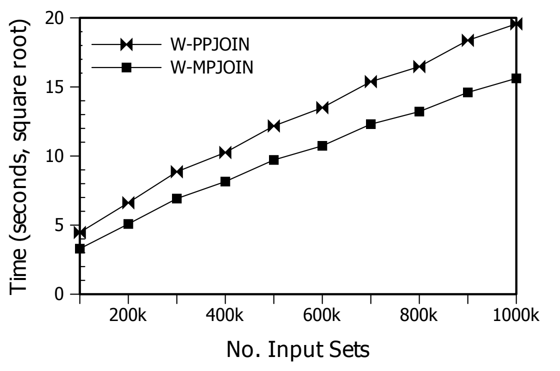
The experiments presented so far established the performance advantages of our algorithms for a wide range of input parameters and dataset characteristics. To demonstrate the generality of our technique, we used datasets derived from string data—on which set similarity join has been the method of choice to realize similarity matching based on q-grams [CGK06, AGK06]; we also employed synthetic datasets in order to flexibly obtain different data distributions. As we will show, datasets derived from XML data using the tokenization functions presented in the previous chapters exhibit similar characteristics to those derived from string data or to the synthetically generated data. The results on XML data regarding the relative performance between the algorithms followed identical trends to those reported in the previous experiments;



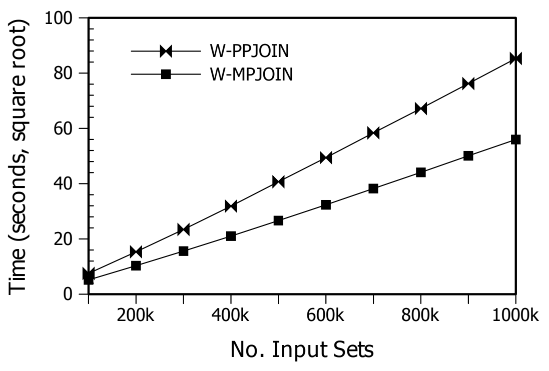
(a) Synthetic dataset, default parameters, unweighted sets



(b) DTA dataset, unweighted sets



(c) Synthetic dataset, default parameters, weighted sets

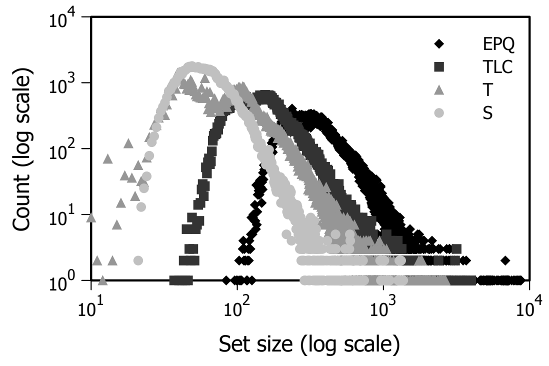


(d) DTA dataset, weighted sets

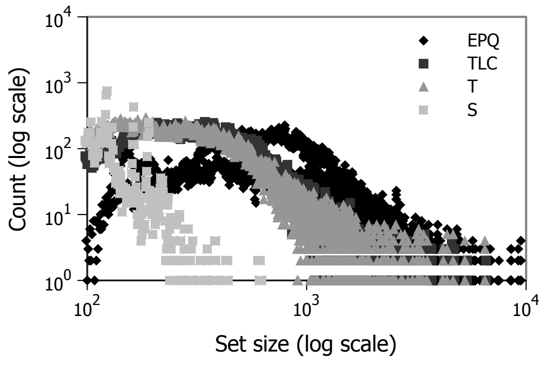
Figure 5.8: Scalability results using synthetic and real datasets

hence, we omit these results for brevity.

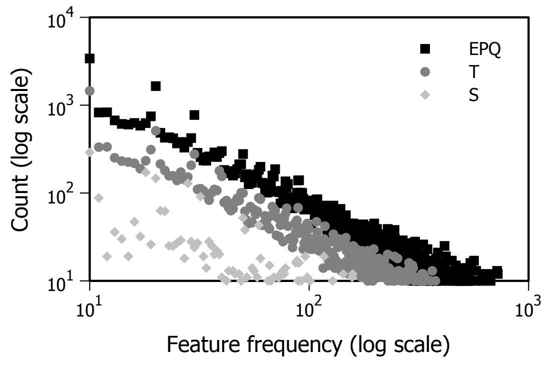
In this section, we quantify the runtime performance achieved by our algorithms using different XML similarity functions, processing strategies, and varying thresholds. We consider the similarity functions described in Chapter 4, i.e, EPQ, PCL_{tlc} (abbreviated in the experimental charts to TLC), and PCL_{slc}. For EPQ, we derive sets from *epq*-v2 profiles, which showed best accuracy results, and $p = q = 2$. For PCL_{tlc} and PCL_{slc}, we used $q = 2$. Further, we used the multi-set representation described in Section 5.6 for PCL_{slc} and consider two processing strategies: SLC-t, which uses the sets derived from $pcl[t]$ as primary sets and SLC-s, which uses the sets derived from $pcl[s]$; in both strategies, we used uniform similarity weights, i.e, $\lambda_p = \lambda_s = 0.5$. We report runtime performance results with varying threshold parameter; for SLC-t and SLC-s, the threshold parameter corresponds to the value of τ_p. As in the accuracy experiments presented in Section 4.5, we used the IDF weighting scheme and Jaccard; the set similarity join algorithm is *w-mpjoin*.



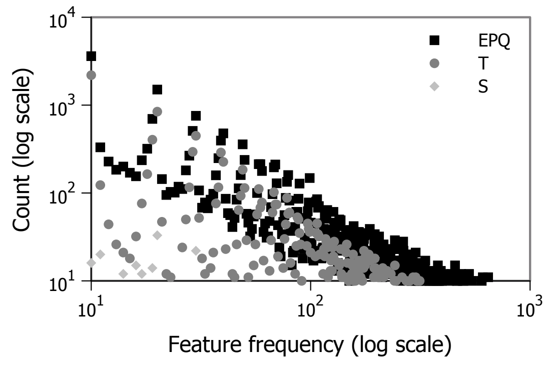
(a) Set size distribution, SwissProt dataset



(b) Set size distribution, PSD dataset



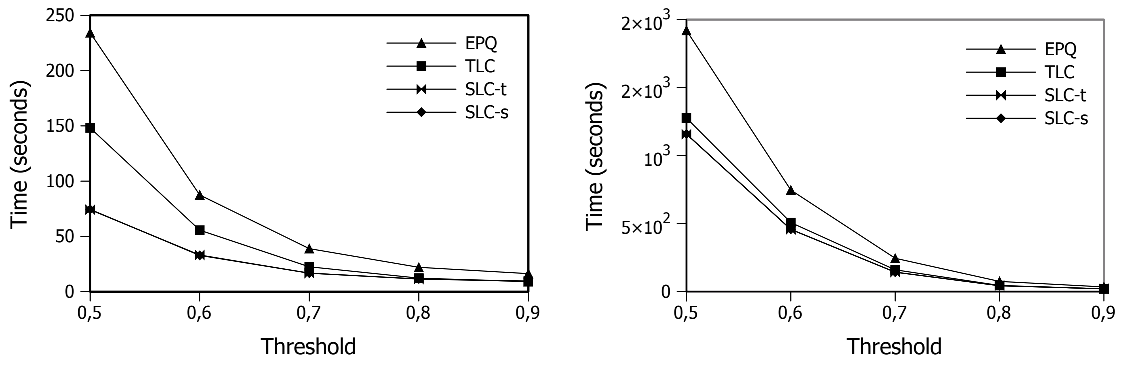
(c) Count-frequency plot of tokens, SwissProt dataset



(d) Count-frequency plot of tokens, PSD dataset

Figure 5.9: Set size and and token frequency distributions of XML data

For this experiment, we used fuzzy copies of the *SwissProt* and *PSD* datasets, each copy containing 100k subtrees. The generation parameters were the same as for the datasets *SwissProt-Pt2* and *PSD-Pt2* (see Table 4.2) with 50% of error extent, i.e., moderate error dataset. As for string data, we used the Karp–Rabin fingerprint function to map tokens to hash values, ordered the tokens within a set according to their frequency, and stored the sets in ascending size order. The resulting datasets exhibit very different characteristics. Figures 5.9(a) and 5.9(b) plot the set size distribution for *SwissProt* and *PSD* datasets, respectively. In the charts, *T* refers to the size of the sets derived from $pcl[t]$ and *S* to the sets derived from $pcl[s]$. Similar to string data, the set size distribution of the *SwissProt* datasets closely follow a log-normal distribution; on the hand, the set sizes of the *PSD* datasets are more spread out. Figures 5.9(c) and 5.9(d) show the count-frequency plot of the datasets. As the TLC distribution is very similar to T, we omitted it in both charts to avoid clutter. The distributions of EPQ and T on *SwissProt* (Figure 5.9(c)) resemble a power-law relationship, which suggests that tokens containing textual information dictates the shape of the distributions, whereas tokens from S concentrates around (relatively) lower frequencies. The count-frequency plot of all datasets derived from *PSD* (Figure 5.9(d)) is quite dif-



(a) Runtime, SwissProt dataset (b) Runtime, PSD dataset

Figure 5.10: Performance results on XML data with varying threshold

ferent those from *SwissProt*. In particular, we observe that they exhibit much lower incidence of low-frequency tokens.

Figures 5.10(a) and 5.10(b) show the runtime results. *EPQ* is slower than *TLC* by a factor of up to 1.8. This result reflects the average token-set size produced by each approach, as *EPQ* delivers larger sets than PCI-based tokenization functions on both datasets. Accordingly, *SLC-t* and *SLC-s* are the fastest because the set similarity join is executed on smaller (primary) sets (i.e., on *T* or *S*). Note however that the results of *SLC-t* and *SLC-s* are based on the reduced threshold τ_p. This means that, for example, while the similarity join is executed with, say, $\tau = 0.8$, on EPQ and TLC, on *SLC* datasets it would be executed with $\tau_p = 0.6$ (recall that we set $\lambda_p = \lambda_s = 0.5$). Taking into account the threshold reduction, both *TLC-t* and *TLC-s* are the slowest; for example, compare their results at $\tau_p = 0.6$ with those of EPQ and TLC at $\tau = 0.8$). Further, the runtime performance of *SLC-t* and *SLC-s* are practically identical. This fact illustrates a trade-off between set sizes and skewness: while *TLC-s* executes the set similarity join on smaller structural sets (i.e., *S*), the primary sets used in *TLC-t* are more skewed. (i.e, *T*). Moreover, this result substantiates our observation that selectivity has low influence on the overall runtime performance: although similarity predicate on tokens containing textual information are expected to be more selective than those on structural tokens, $SLC\text{-}t$ is not faster than $SLC\text{-}s$. Finally, even though the *PSD* datasets exhibit more spread set size distribution, which favors our min-prefix algorithm, the results on *SwissProt* are about one order of magnitude faster as compared to *PSD*. The reason is that the *SwissProt* has much more incidence of low-frequency tokens than *PSD*. As already discussed, low-frequency tokens translates into shorter inverted lists and much less prefix overlap between dissimilar sets.

5.7.7 Experimental Summary

In all measurements performed on synthetic and real datasets, *mpjoin* and *w-mpjoin* provided a superior performance than their competitors. We have shown that a large part of the sets processed during the candidate generation phase are indeed associated with stale tokens, i.e., these sets only add unnecessary overhead, because they cannot be part of the final result. The optimizations in the verification phase turned out to be effective: the runtime did not blow up even with a dramatic 18x increase of the number of candidates. As opposed, suffix filtering was ineffective: the reduction of the candidates did not compensate the increased runtime in the candidate generation phase—even when employed together with the min-prefix technique. This fact emphasizes our observation in Section 5.2 that reduction of the candidates should be considered with care, because it does not always translate into performance gains.

The spread of the set size distribution has more impact on the performance than the set size itself. The algorithms run faster on datasets exhibiting highly dispersed set sizes than datasets containing more uniform set sizes. In particular, the performance gain of *mpjoin* and *w-mpjoin* increases with the set size variance, because tokens associated with indexed sets became stale more quickly in the course of processing. The performance increases with the skew of the token frequency distribution because inverted lists become shorter. On the other hand, the advantage of using the min-prefix technique to minimize of length of the inverted lists diminishes. By varying input size as well as the threshold parameter, both *mpjoin* and *w-mpjoin* steadily revealed their performance advantage. The runtime of all algorithms grows quadratically as the input set increases, as expected. Finally, on XML data, executing our algorithms using PCL_tlc as similarity function is about 1.8 faster than using EPQ; for PCL_slc, we observed identical results using the structural and textual sets as primary sets.

5.8 Related Work

There is a vast body of literature on performing similarity joins in vector spaces; in this context, a similarity join is a variant of the more general approach known as *spatial join*. See [JS07] for a recent survey. Indexing techniques for vector spaces are well-suited for implementing similarity joins in application domains where the objects can be described by low-dimension token vectors and the notion of similarity can be expressed by a distance function of the Minkowski family, such as the Euclidean distance (L_2 norm). However, all these techniques suffer from the "curse of dimensionality", meaning that their performance degrades as the underlying dimensionality increases; they are often outperformed by sequential scans for more than around 10 dimensions. As already mentioned, although data reduction techniques can be used to uncover the intrinsic dimension, for domains involving text data, the dimensionality of the reduced dataset is however expected to be high enough to rule out spatial join methods (see also discussion in Section 2.1.2).

In many applications, the objects of interest cannot be properly represented as a collection of features (tokens, in our case) or distance functions of the Minkowski family are not appropriate. An alternative method for such situations is to resort to generic metric spaces where the only information available is the pairwise distance between objects given by a *distance metric* [CNBYM01]. In this context, a common approach consists of using embedding methods to map object from the original metric space into a vector space; afterwards, spatial indexes can be used to save distance calculations. See [JLM03] and [GJK$^+$06] for such approaches on text and XML documents, respectively. Other techniques reduce the dependency on the parameter space by avoiding the use of pre-built indexes. For example, reference [JS08] presents the Quickjoin algorithm, which recursively partitions the data until each partition is sufficiently small such that nested-loop joins can be applied in an economical way. Quickjoin has been shown to perform better than competing methods that execute likewise without index support. Note that the performance of all metric-space methods heavily relies on a good pivot selection strategy. Unfortunately, this problem is not well understood and, in practical applications, pivots are randomly chosen even though it is likely to lead to suboptimal results [CNBYM01].

Set similarity joins embody an important method to implement similarity joins when the objects of interest lend themselves to set representation and exhibit high dimensionality, such as text data (see other examples of application areas at the beginning of this chapter). In this context, a rich variety of optimizations has been proposed—most of them were discussed in Section 5.1 and are used by our algorithms: derivation of bounds (e.g., size bound [SK04, AGK06, BMS07, XWLY08, TSP08]), exploitation of a specific data set order [SK04, BMS07, XWLY08] and token positional information [XWLY08], and signature schemes [AGK06, CGK06]. The prefix-filtering concept was first exploited to improve set similarity joins in [SK04] and formally defined in [CGK06]; in these contributions, prefix-filtering is used without any information concerning set sort order and therefore corresponds to our definition of *maxprefix*. The use of smaller prefixes for indexing, i.e. *midprefix*, was first employed in [BMS07] and formally defined in [XWLS09].

There are two main query processing models for set similarity joins. The first one uses an *unnested* representation of sets in which each set element is represented together with the corresponding object identifier. Here, query processing is based on signature schemes and commonly relies on relational database machinery: equi-joins supported by clustered indexes are used to identify all pairs sharing signatures, and grouping and aggregation operators together with user-defined functions (UDFs) are used for verification [CGK06, AGK06]. The second model is related to Information Retrieval techniques. An index is built for mapping tokens to the list of objects containing that token [SK04, BMS07, XWLY08]. The index is then probed for each object to generate the set of candidates which will be later evaluated against the overlap constraint. Previous work has shown that approaches based on indexes consistently outperform signature-based approaches [BMS07] (see also [HCKS08] for selection queries). As primary reason, a query processing model based on indexes provides superior optimization opportunities. A major method for that uses an index reduction technique such as prefix-filtering [BMS07, XWLY08], which is also the main focus of this chapter. Furthermore, most signature schemes are *binary*, i.e., a single shared signature suffices to elect a pair of sets as candidates. Also, signatures are solely used to find candidates; matching signatures are not leveraged in the verification phase. As a result, both sets representing a candidate pair must be scanned from the beginning to compute their similarity. In contrast, approaches based on indexes accumulate overlap scores already during candidate generation. Hence, set elements accessed in this phase can be ignored in the verification. Inverted lists have also been shown to perform better than signature-based approaches on simpler kinds of set joins, such as strict containment and non-zero overlap joins [Mam03].

In addition to exact optimizations, dimension reduction methods can be used to speed-up set similarity join processing at the cost of producing approximate results (some valid object pairs may be missed). LSH is the most popular technique for approximate similarity joins: it can be used to reduce the size of the input sets [Bro97] as well as signature schemes [CDF$^+$01]. Not all set similarity functions admit an LSH hash function family however. A required property is that the distance formulation of a similarity function Sim, i.e. $1 - Sim$, must satisfy the triangle inequality [Cha02]. For example, Jaccard admits an LSH hash function family, whereas there is not such a function for Dice [Cha02].

Text similarity joins were first proposed in the context of data integration in the seminal work of Cohen [Coh98]. In this regard, SED is typically used as the similarity join predicate. A common approach consists of mapping strings to sets of q-grams and using set-overlap as edit distance constraint. The work in [GIJ$^+$01] implements this approach on top of a relational query engine. Xiao et al. [XWL08] exploit the locations and content of mismatching q-grams to improve filtering.

A related line of work focuses on set-similarity selection queries. The work in [HCKS08] exploits size bounds of the IDF similarity and token ordering in inverted lists to design highly efficient algorithms. Li et al. [LLL08] propose algorithms to optimize the merging of inverted lists, which is used to calculate the overlap score of candidates, and investigate different filtering configurations.

Recent trends concerning similarity joins include: exploitation of parallelism and sort and searching capabilities of GPUs [LSS08]; compact representation of similarity join results [BEF08]; and text similarity joins using string transformations [ACK08]. Moreover, following the idea presented by Chaudhuri et al. [CGK06], recent work addresses the integration of similarity joins into DBMS query engines. For example, [LNS09] exploits Power-law distributions to estimate the sizes of set similarity join outputs and [SAA10] introduces transformation rules for the optimization of logical query plans containing similarity join operators.

5.9 Summary

In this chapter, we proposed a new index-based algorithm for set similarity joins. Following a completely different approach compared to previous work, we focused on a reduction of the computational cost for candidate generation as opposed to lowering the number of candidates. For this reason, we introduced the concept of min-prefix, a generalization of the *prefix filtering* concept, which allows to *dynamically* and *safely* minimize the length of the inverted lists; hence, a larger number of irrelevant candidate pairs is never considered and, in turn, a drastic decrease of the candidate generation time is achieved. As a side-effect of our approach, the workload of the verification phase is increased. Therefore, we optimized this phase by stopping as early as possible the computation of candidate pairs that do not meet the overlap constraint. Moreover, we improved the overlap score accumulation by storing scores and auxiliary information within the indexed set itself instead of using a hash-based map. Finally, for dealing with massive datasets, we presented variants of our algorithm for disk resident data and parallel execution. Our experimental results on synthetic and real datasets confirm that the proposed algorithm consistently outperforms the known ones for both unweighted and weighted sets. Finally, we identified characteristics of the input dataset that influence the performance of set similarity join algorithms and compared the performance delivered by the similarity functions presented in the previous chapters.

Chapter 6

Integration into XTC

So far, our discussion has mostly been "system-oblivious" in the sense that we have generally referred to the input to our tree similarity join (TSJ) algorithms as XML collections or datasets and abstracted from details of the underlying data server system. In practice, however, mere operating system's file management is rarely enough for supporting data-intensive applications over large XML collections; instead, as for most kinds of data, database management systems are needed to free up XML application programs from dealing with concurrency and failures while providing efficient data access and maintenance, among other benefits. In this regard, our goal in this chapter is to investigate several aspects of the integration of our TSJ operators into a native XML database management system (XDBMS).

There has been a myriad of proposals for the support of XML data in DBMSs. Early work was characterized by a clear distinction between two approaches: fully leveraging traditional relational database engines by splitting XML documents for storage into multiple columns and tables (shredding) and using algorithms to map XML queries onto SQL queries and convert relational results to XML—such functionalities are encapsulated in a thin layer on the top of the architecture, e.g., see [STZ+99]—,and implementing from scratch tailor-made XDBMSs, e.g., [MAG+97][1].

Following a common course of DBMS architectural evolution, the above distinction has faded with the arise of the so-called hybrid systems, which promote the co-existence of relational and XML data by providing deeper integration and transparent access to XML functionality [BCJ+05, MLK+05, Rys05]. The most significant extensions to the conventional relational systems are: support to SQL/XML [sql03] as well as XQuery in a unified query model; XML data abstraction via the XMLType, a new data type defined by the SQL/XML standard; augmentation of the logical and physical algebra with new XML constructs—along with the respective optimization rules and cost models—represented by the same internal data structures as ordinary SQL constructs, thereby allowing reuse of the existing query optimizer framework; new indexing mechanisms for XML content, structure, or both (using B-trees and auxiliary tables); and storage models based on XML tree partitioning or serialized

[1]Initially proposed in the context of semi-structured data in general.

(binary) formats. In this connection, the work in [LCBC08] presents a query optimization framework supporting multiple storage and indexing models.

Despite of only supporting XML data and, hence, having a narrower scope, XDBMS architectures still have their merit not only in providing data management to XML applications but also in pushing the boundaries of DB research on XML in general. In fact, several techniques adopted by commercial hybrid systems were first proposed in the XDBMS environment. See [Mat09] for a comprehensive study involving state-of-the-art XDBMS solutions as well as related discussions on techniques developed in the context of relational and hybrid systems.

Here, we conduct our work using the XML Transaction Coordinator (XTC) as XDBMS platform [Hau05, Mat09, xtc]. Although being a research prototype, XTC implements almost all features of a full-fledged XDMS including: multi-lingual query interfaces (XQuery, XPath, DOM, and SAX); an extensible, rule-based, and cost-based[2] query optimization framework supporting a rich repertoire of logical and physical algebraic operators; cardinality estimators; a variety of index structures; node-oriented and path-oriented storage models; logging and recovery services; and support to new hardware architectures such as tailored buffer management algorithms for flash disks. As a unique feature, XTC provides fine-granular transaction isolation mechanisms allowing read/write transactions and collaborative XML document sharing with ACID guarantees.

As already pointed out in the first chapter, we view the embedding of similarity join operators into a database engine as belonging to the broader context of DB/IR integration. The increasing appeal for DB/IR architectures comes from the urgent need—fueled by emerging classes of applications, in particular, Web-based applications—of seamless management of a full spectrum of data, ranging from totally structured, semi-structured, to totally unstructured data. Along this range, the operations and underlying methods for querying the data move from DB to IR: transactionally protected, structured, Boolean queries based on the exact matching paradigm (DB world) to read-only, keyword-based queries following the similarity matching paradigm and producing ranked results (IR world). In this context, the similarity join is situated between these two worlds: a typical DB operation for combining data defined under the similarity matching paradigm. Furthermore, because a very common XML use case is the representation of semi-structured data, support of similarity matching or IR-style query processing is an essential feature for fully realizing the value of XML [TW01]. In fact, the XML data model can be seen as the natural bridge between DB and IR technologies. As mentioned above, XTC already provides most of the functionalities found in traditional DB systems. The work described in this chapter can be viewed as an initial step towards making XTC a DB/IR architecture.

There are many challenges for incorporating similarity matching operators into database systems (or building DB/IR systems, in general) [CRW05, Wei07]. A not exhaustive list of challenges includes: marrying the semantics of structured queries (unordered and complete set of results) with unstructured queries (ranked and incom-

[2]Cost-based optimization is not fully operational yet.

plete results) into a (not exceedingly complex) formal query model and related algebra; optimization and execution of queries combining exact and similarity matching paradigms as well as queries on both structure and text; integration of relational indexes (e.g., B-trees) with inverted-list-based indexes into a common indexing model; transactional index maintenance; flexible ranking and scoring; exploitation of taxonomies or ontologies for improving query answering; and data transformation—note that these challenges, especially the latter three aspects, are also pertinent in the context of EM systems, especially in scenarios demanding on-the-fly similarity matching such as EII or dataspace systems. In the light of these challenges, it is not a surprise that most commercial products providing DB/IR integration are either highly specialized vertical solutions or loosely coupled DBMS and IR engine systems. In the latter, special language predicates are used to guide the optimizer in routing parts of the query to one of the two engines—with very little scope for optimization—and the results of both of them have to be combined at a separate integration layer before being returned to the user.

In this chapter, it is not our aim to encompass all aspects around the integration of our similarity join framework in XTC. As already made clear, despite of being aggressively pursued in the recent years, tight integration of DB and IR technologies is still a long-term goal, which will require from both research communities a great deal of (ideally collaborative) work. Instead, we focus on exploiting XDBMS-specific features to optimize the realization of the similarity joins algorithms proposed in this thesis. Specifically, we leverage XTC's storage model, indexing infrastructure, node identification mechanism, and physical algebra for XQuery processing to locate qualified XML fragments, to efficiently generate XML tree representations, and to compose pipelined query evaluation trees. As a result, TSJ operators can be embedded into regular XML queries to deliver more complex EM solutions. Later in this chapter, we will discuss the flurry of recent work addressing many aspects DB/IR integration, which can be used to promote further integration in our context.

The remainder of this chapter is structured as follows. In Section 6.1, we provide an overview of the XTC architecture highlighting the most important elements for our study. In Section 6.2, we present the TSJ operator, the principal component of our framework and outline the sequence of steps to derive a similarity join result. We discuss strategies to fetch subtrees to be compared from their disk-based storage locations to a memory-resident working area in Section 6.3. Next, in Section 6.4, we describe the profile generation for ordered (*epq*-grams tokens) and unordered trees (PCI-based tokens), which leverage XDBMS internals for faster processing. Construction and maintenance of auxiliary data structures are discussed in Section 6.5. The composition of physical query plans that leverage existing XML query processing algebra is covered in Section 6.6. Future work on providing deeper integration of our framework is discussed in Section 6.7 and experimental results validating our approach are presented in Section 6.8. We discuss related work in 6.9, before we wrap up with a summary in Section 6.10.

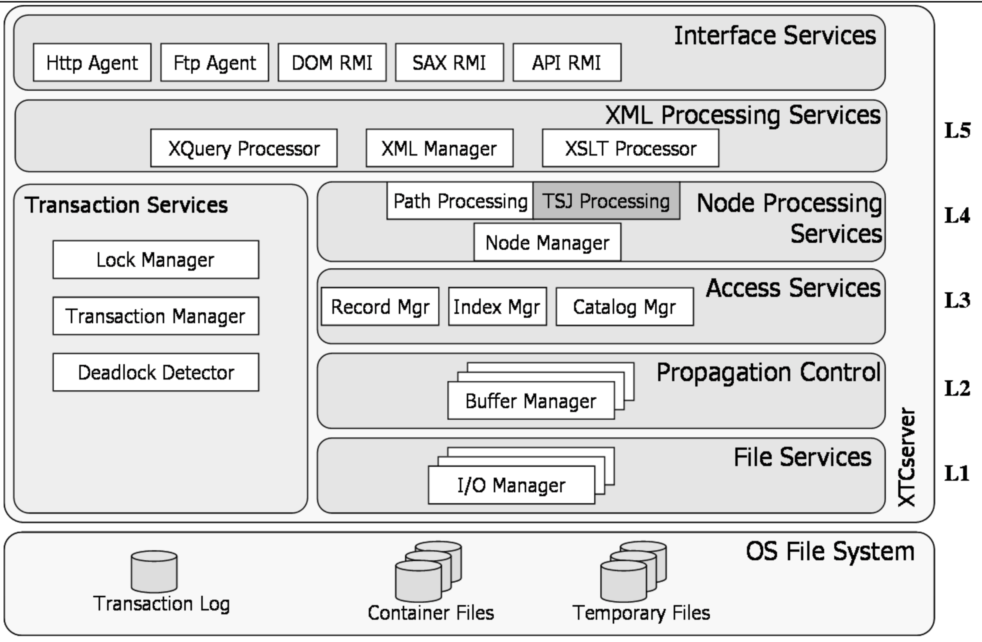


Figure 6.1: The five-layer architecture of XTC including the component for tree similarity join (TSJ) processing in the fourth layer

6.1 The XML Transaction Coordinator

In this section, we first sketch the major concepts of the multi-layered architecture of XTC and, then, review some internal components of XTC. Our discussion is limited to the components that we exploit to optimize our similarity join algorithms, namely: node identification scheme, path-oriented storage model, and index structures. We refer the reader to [Hau05, Mat09] for a complete and detailed description of XTC, which cover all aspects missed here.

6.1.1 XTC's Architecture

The design of XTC adheres to the classical five-layer architecture introduced by Härder and Reuter [HR83]. Figure 6.1 illustrates how XTC is structured according to this reference architecture. The decomposition of XDBMS functionality into five layers provides a mapping model enabling dynamic abstraction through the entire flow of data within a DBMS: from the level of non-volatile storage devices up to the user interface. In this context, we accommodate the TSJ processing functionality on top of the storage engine, i.e., in the fourth layer. Note that this approach corresponds to the *RISC* strategy for DB/IR integration, which is postulated by Chaudhuri et al. in [CRW05].

The two bottom-most layers, i.e., *File Services* layer (L1) and *Propagation Control* layer (L2), are responsible for the management of external storage and DB buffer, respectively, and have basically the same characteristics as in relational systems. All data (payload, metadata, and auxiliary data) is stored on disk in container files, which are plain files internally arranged in a sequence of fixed-sized *blocks*. Each container is associated to an *I/O-Manager* at L1, which provides read/write block operations between container files and main memory. The next higher layer, i.e, L2, implements page-oriented mapping to external storage by designating a buffer manager to each I/O-Manager. Each buffer manager maintains an array of fixed-size *page frames*, whose size is the same as the container blocks. Page requests from the higher layer are handled by standard displacement algorithms (e.g., LRU) and fix/unfix mechanisms.

The *Access Services* layer (L3) implements the mapping of XML trees to the pages managed by L2 and several data access methods, including indexes and tree scan methods. Also, metadata about stored documents is managed at this layer. The components in charge of providing these functionality are *Record Manager*, *Index Manager*, and *Catalog Manager*. This layer defines the storage model of XTC. Currently, two models are supported in XTC: node- and path-oriented storage models. In the former model, each node is encoded and mapped onto external memory, while, in the latter, inner nodes (structural part) are *virtualized* with support of a PS structure. The algorithms presented in this chapter are applicable to the path-oriented storage model. Hence, we focus on this model in the upcoming sections.

Upwards in the hierarchy, we have the *Node Processing Services* layer (L4). The *Node Manager* provides an interface for operations such as tree traversal, tree scan, and navigational primitives based on the DOM operations (e.g., *getParentNode* and *getLastChild*), decodes nodes represented in internal record format to the external format, i.e., human readable XML nodes, and issues lock requests to the *Lock Manager* according to the respective operation and isolation level. XTC's physical algebra for XQuery processing is contained in the *Path Processing* Component. All physical operators are implemented as *iterators* [Gra93], i.e., their interface consist of the methods open, next, and close (the so-called ONC protocol). We have extended this algebra with similarity-specific operators (*TSJ Processing*); we defer details about this extension to Section 6.6.

Transaction Services span all the previous layers. *Lock Manager* and *Deadlock Detector* are responsible for concurrency control (implemented by a (pessimistic) locking protocol), while log and recovery is provided by the *Transaction Manager*. To increase potential parallelism, XTC adopts a logical internal representation of XML documents based on an extension of the DOM data model, the so-called *taDOM* (see [Hau05] for details).

The upmost layer embodies the *XML Processing Services* layer (L5), which represents the non-procedural interface of XTC. In particular, L5 provides evaluation and optimization of XQuery expressions (XQuery Processor). Complementing the five-layer architecture, *Interface Services* support several communication protocols allowing to link XTC to specific programming environments. In particular, XTC executes DOM calls internally, instead of reconstructing and transferring the entire document to the client (as is done by most XDBMSs). Currently, our similarity join framework is neither supported at L5 nor exposed to client applications by the Interface Services.

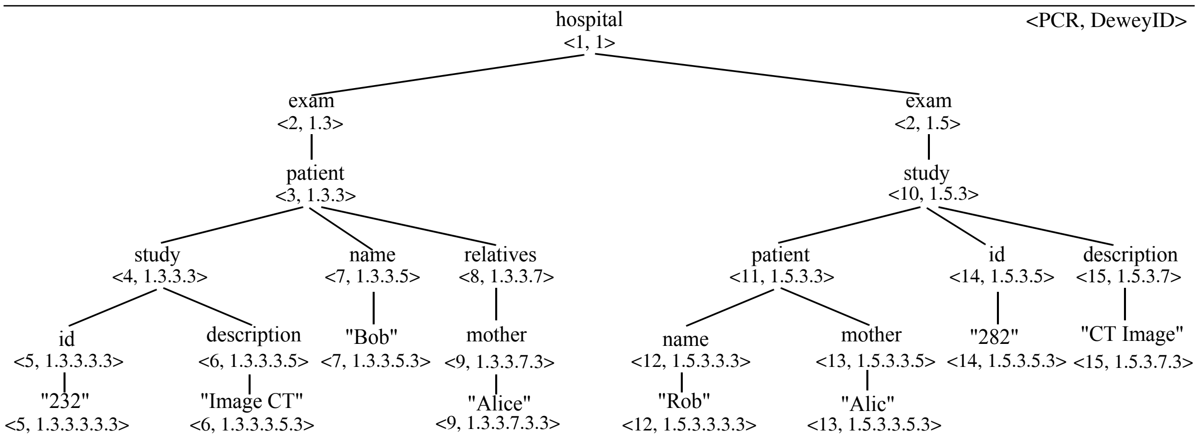


Figure 6.2: XML document fragment from Figure 2.4 identified with DeweyIDs (related PCR values also shown)

Instead, we provide an access module which allows calling the TSJ operator directly at L4.

6.1.2 Node Identification using DeweyID

In relational systems, *record identifiers* (RID) specify the record's page number and offset within the page and, hence, uniquely identify the corresponding records. RIDs play an important role for query processing, for example, in queries involving conjunction of predicates on a table: index scans for each predicate are performed to obtain sorted RID lists that are intersected via an AND-tree. This strategy allows carrying out most of the processing without needing to fetch the actual records from the physical storage.

A similar approach can be employed for XML query processing by using node identifiers. However, only equality comparisons are insufficient, because navigational and declarative XML operations mostly involve comparison of nodes in terms of their relationships—of particular importance are the axes *parent/child*, *ancestor/descendant*, *previous-sibling/following-sibling*, and *previous/following*. Other frequent operations are sorting of nodes according to the document order and identification of the nesting level of a node. Besides query processing, quick determination of a node's ancestor chain is instrumental for the realization of fine-grained locking protocols [HHMW07]; and for keyword query processing [XP05], a key operation is the computation of the *lowest common ancestor* (LCA) for a set of nodes (in the next section, we will see that efficient LCA computation is also an important requirement for the path-oriented storage model of XTC).

The above requirements emphasize the need for a structure-aware node identification mechanism. In this context, prefix-based identification schemes, such as

DeweyIDs [HHMW07] and ORDPATH [OOP$^+$04], are already standard. XTC uses DeweyIDs for node identification (ORDPATH or any other prefix-based scheme would also be appropriate as they provide equivalent functionality).[3] A DeweyID encodes the path from the documents' root to the node as well as sibling order. DeweyIDs are represented by a sequence of integers called *divisions* (separated by dots in the external format). Figure 6.2 shows the XML tree from Figure 2.4 identified with DeweyIDs (disregard the value called PCR for the moment).

Remarkably, DeweyIDs capture a large part of the structural information of XML nodes: all structural relationships between two nodes can be inferred; identification of the node's ancestor ID list (for example, in Figure 6.2, the ancestors IDs of the text node with value “Alice” and DeweyID 1.3.3.7.3.3 are 1.3.3.7.3, 1.3.3.7, 1.3.3, 1.3, and 1; the lexicographic order of DeweyIDs corresponds to the document order of the respective nodes; the level of a node is given by the number of divisions of its DeweyID minus one; and the DeweyID of the LCA for a set of nodes can be obtained by calculating their longest common DeweyID prefix.[4]

Further, existing DeweyIDs are immutable, i.e., they allow the assignment of new nodes and IDs without the need to reorganize the IDs of existing nodes. When identifiers deprave after weird insertion histories, re-identification can be preplanned; it is only required, when implementation restrictions are violated, e.g., the max-key length in B*-trees. To avoid node re-identification as much as possible, Härder et al. [HHMW07] employed a *dist* parameter to increment division values thereby leaving gaps in the numbering space between consecutive identifiers (in Figure 6.2, the value of the *dist* parameter is 2) and complementary overflow mechanisms when gaps for new insertions are in short supply. Finally, although DeweyIDs may become arbitrarily large depending on the nesting level or update activity, prefix compression techniques can reduce the space consumption to about 25% of the uncompressed format [HHMW07].

6.1.3 Path-oriented Storage Model

We have seen that the DeweyID mechanism allows inferring many structural aspects of a given node. In particular, given a DeweyID, we can obtain its list of ancestor IDs by removing division values in a step-wise way. However, DeweyIDs do not provide node label information. This means that, to derive the path from the root node to

[3] Note that node identification mechanisms in XML are commonly referred to as *node labeling schemes*; in particular, the term SPLID (Stable Path Labeling IDentifier) has been used to designate prefix-based mechanisms such as DeweyID and ORDPATH in the XTC literature [HMS07]. However, we refrain from using this nomenclature because we have already been using the term “label” to refer to the second element in the triple defining XML nodes (see Section 2.2.1), aka node tag or node name.

[4] The DeweyID encoding scheme adopted by XTC is slightly different from the one depicted in Figure 6.2: special division values are used for strings and attributes as well as other minor modifications to accommodate the extensions of the taDOM data model [Mat09]. Because these modifications are irrelevant in the context of our simplified XML data model (e.g., attribute nodes are treated as element nodes), and consequently, not important for our discussion, we ignore them here.

PCR-PCI Table

PCR	PCI
5	1
6	2
7	3
9	4
12	3
13	4
14	1
15	2

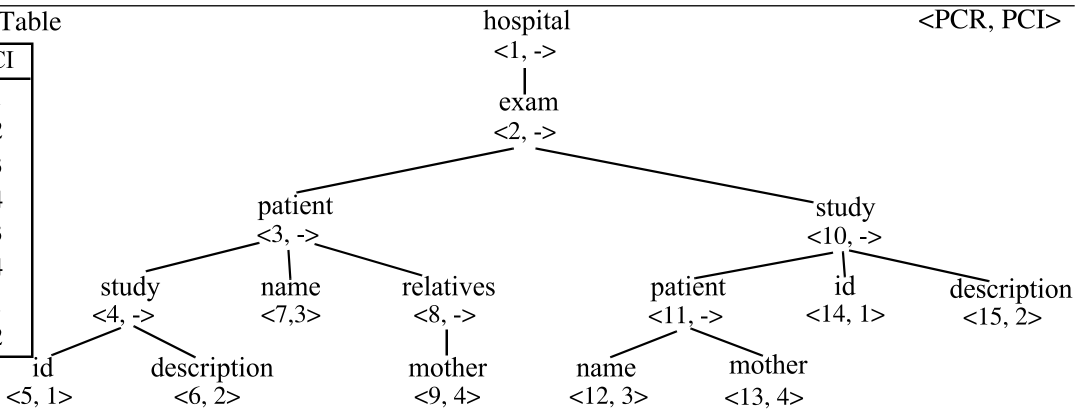


Figure 6.3: PS using PCRs to identify paths to the root (related PCI values and PCR-PCI table are shown)

a given node, access to the storage system is nonetheless necessary. In Chapter 3, we have presented the PS structure, which describes all path classes from an XML collection. In this connection, the complementary nature of DeweyIDs and the PS is evident: by associating the DeweyID of a node with the corresponding node in the PS, we are able to derive not only the label of this node, but also its complete path information. In order to do so, we need to simply number all nodes in a PS, thereby uniquely identifying all (partial) paths appearing in an XML database. This identifier is referred to as Path Class References (PCRs) [HMS07]. Figure 6.3 illustrates the PS structure relative to the XML tree depicted in Figure 6.2, with all the nodes identified by PCRs values. (Note that leaf nodes are associated with PCI values. This association is represented in the *PCR-PCI table* on the left-hand side, whose utility will be explained shortly.)

Consider again Figure 6.2. Note that element nodes are associated with the respective PCR in the PS; text nodes are associated with the PCR of its owing element. The PCR value for the text node with value "Alice" and DeweyID 1.3.3.7.3.3 is 9. Using the DeweyID in conjunction with the PS structure, we can now reconstruct the path from the root to this node, therefore obtaining $\langle$(hospital,1),(exam,1.3),(patient,1.3.3), (relatives,1.3.3.7), (mother,1.3.3.7.3)$\rangle$.

The ability of reconstructing paths instigates the design of a space-economic storage model: one can store only the content part of XML documents, i.e, their text nodes values, together with the corresponding DeweyID and PCR; as a consequence, the structural part is virtualized and can be reconstructed whenever needed—when the last element of a path is an empty element node, a null node is stored together with the PCR of the element node. This strategy for storing XML documents in XTC is referred to as the *path-oriented storage model*—contrast this model with the node-oriented storage model, where all nodes, structural and textual, are stored on disk. As already mentioned in Chapter 3, most real-world XML data exhibit highly repetitive

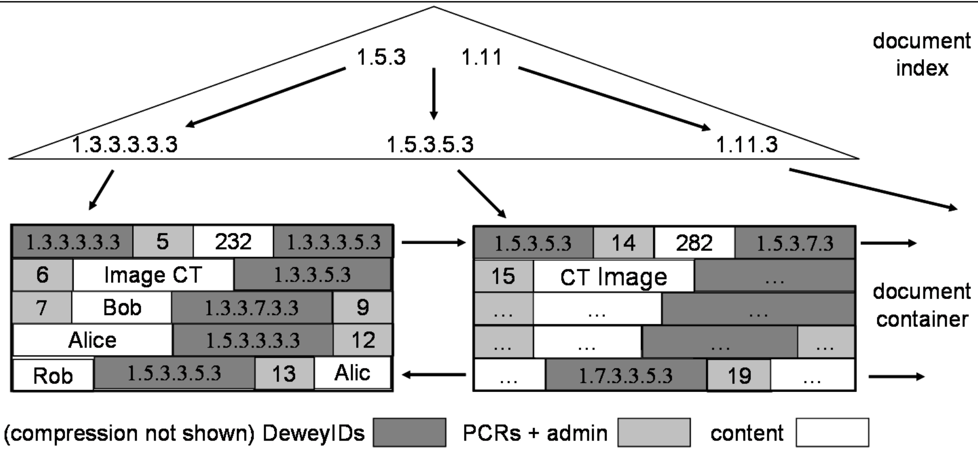


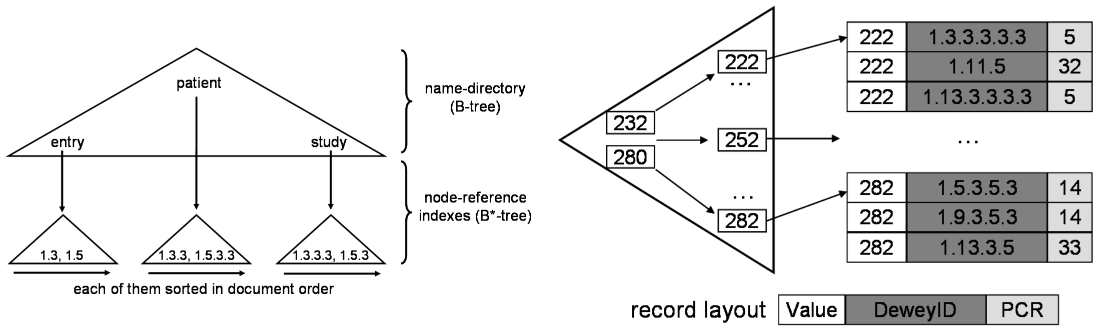
Figure 6.4: Stored document in path-oriented format

structure, where several path instances reappearing many times in an XML collection. Hence, path-oriented storage can achieve substantial space savings: from 20% to 70% as compared to the size of the plain textual representation of documents; and from 5% to 30% as compared to node-oriented storage[5].

Figure 6.4 zooms in on the path-oriented format. The so-called *document index*, a B-tree with key/pointer pairs $\langle DeweyID{+}PagePtr \rangle$, indexes the first node in each page of the document container, which consists of a set of doubly chained pages. Several page types can be assigned to enable allocation of pages for documents, indexes, etc., in the same container. Further, using sufficiently large pages—varying from 4K to 64K bytes, which can be configured to the document properties, the document index is usually of height 1 or 2. Because of reference locality in the B-tree while processing XML documents, most of the referenced tree pages are expected to reside in DB buffers—thus reducing external accesses to a minimum. In each page, text nodes of variable length are stored in document order, each node composed of entries of the form $\langle SPLID, PCR, value \rangle$.

The performance of operations such as tree reconstruction (or tree scan) and navigation does not degrade with the structure virtualization. In fact, significant performance gains have been observed on several XML documents [Mat09]. In particular, we rely on tree scans to fetch the input to TSJ. Briefly, a tree scan on path-oriented storage is executed in L3 as follows. First, a lookup on the document index is performed using the DeweyID of a given tree root node. The page address and offset returned

[5]Both path- and node-oriented storage employ compression of DeweyIDs, node labels (replacing of elements labels by a two-byte integer values), and content. Prefix-based compression of DeweyIDs is less effective on path-oriented storage, because ordered sequences of DeweyIDs are non-dense due to the absence of the DeweyIDs from structure nodes. Moreover, this storage layout requires extra-bytes per node for storage of PCRs together with some administration data. Nevertheless, structure virtualization pays off in all cases for a wide variety of XML documents [HMS07].



(a) Element index (b) CAS index

Figure 6.5: Index infrastructure of XTC

corresponds to the first text node (or null node) of the tree. Using the DeweyID, PCR and the PS, all nodes in the path from the root node down to the stored node are reconstructed in internal record format (if the stored node is a text node, it is also reconstructed) and added to the result. Next, the chained pages are processed sequentially. For each node, the LCA between this node and the previous node is calculated, and the path from the LCA node down to the current node is constructed and added to the result. The tree scan stops when the first node, which is not a descendant of the tree root node, is found.

The method above describes the scan process of a single subtree. In Section 6.3, we discuss strategies for fetching all input trees for similarity processing.

6.1.4 Indexing

In addition to the document index, XTC supports a rich variety of indexes that can be specified by the database administrator to speed-up certain queries [Mat09]. Using B-trees as foundational structure, indexes can be defined on XML content, structure, or both. Next, we describe the so-called *element index* and *content-and-structure index* (*CAS index*), as we will consider them in further discussions in this chapter.

An element index consists of a *name directory* with (potentially) all element labels occurring in the XML document (Figure 6.5(a)); this name directory often fits into a single page. Each specific element label refers to the corresponding list of DeweyIDs, sorted according to the document order. In case of short reference list, they are materialized in the index; larger reference lists may, in turn, be maintained by a *node reference index* as indicated in Figure 6.5(a). The name directory is implemented by a B-tree while the node reference index is implemented by a B*-tree. For the latter, the index record format only stores DeweyIDs as keys; the corresponding value field is empty. Optionally, we can extend the functionality of the element index by storing the PCR in the field value. The resulting index allows distinguishing all occurrences of an element label in multiple paths.

CAS indexes embody a hybrid access path mechanism capturing content and structure. The content part is specified by a path query, such as "$//study/id[. = 232]$" (point query) or "$//study/[232 < id < 282]$" (range query)—see [Mat09] for a formal definition of the path query. Alternatively, a disjunction of several path queries can be specified to define a so-called *collective heterogeneous* CAS index. A text node content is indexed if its parent is in the result of the path expression. The record field value consists of the corresponding DeweyID and PCR value. In this record layout, keys are sorted in ascending order while record field values are sorted in document order. A B*-tree supporting duplicate keys is used to implement CAS indexes. Figure 6.5b illustrates a CAS index specified by a range query. As for element indexes, one can flexibly define different CAS index variants by modifying the record layout [Mat09], e.g., by concatenating the PCR value to the key value, we obtain a PCR-based clustering of the index (instead of the DeweyID clustering which results from the default record format).

Finally, in all cases, support of variable-length keys and reference lists is mandatory; additional functionality for prefix compression of DeweyIDs is again very effective.

6.2 The TSJ Operator

After having presented the XDBMS environment, we now start our discussion on the integration of our similarity join framework into this context. The discussion puts together all concepts of the previous chapters and will be organized around the main component of the framework, i.e, the tree similarity join (TSJ) operator. In the following, we first expand Definition 2.1 on Page 40 to encompass all elements of the TSJ operator and make their relationship explicit. Then, we provide a quick overview of the TSJ evaluation, before we delve into details in the subsequent sections.

6.2.1 Tree Similarity Join

The underlying operation of TSJ has four parameters: two expressions defining the set of trees to be compared, the token-based similarity function, and the similarity threshold. The token-based similarity function further unfolds three components: the profile generation method (or tokenization function), the weighting scheme, and the set-overlap-based similarity measure; finally, the tokenization function requires a parameter defining content and structure delimitation.

Definition 6.1 (Tree Similarity Join (TSJ))**.** *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two XML databases and $exp(\mathcal{D})$ be an XPath or XQuery expression over a database $\mathcal{D}$. Further, let tok be a tokenization function that, given a set of PCIs PC_t, maps an XML tree T to a profile $tok[PC_t](T)$, ws be a weighting scheme that associates a weight to every element of a given input set, and ss be a set-overlap-based similarity measure. Let sf be the similarity function defined by*

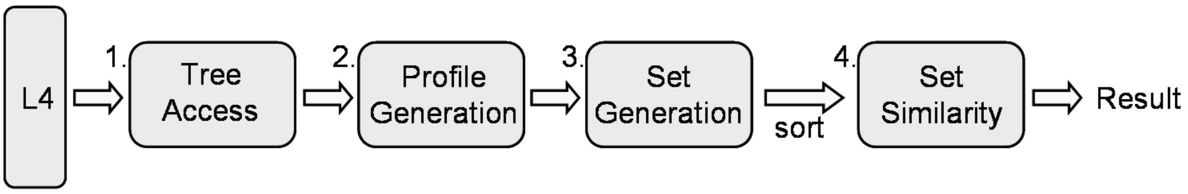


Figure 6.6: Course of TSJ evaluation

the triple $\langle tok[PC_t], ws, ss\rangle$, which returns the similarity between two XML trees T_1 and T_2, $sf(T_1, T_2)$ as a real value in the interval $[0, 1]$. Finally let τ be a similarity threshold, also in the interval $[0, 1]$. The tree similarity join between the tree collections specified by $exp_1(\mathcal{D}_1)$ and $exp_2(\mathcal{D}_2)$, denoted by $TSJ(exp_1(\mathcal{D}_1), exp_2(\mathcal{D}_2), sf, \tau)$, returns all scored pairs $\langle(u_1, u_2), \tau\prime\rangle$ s.t. $(u_1, u_2) \in exp_1(\mathcal{D}_1) \times exp_2(\mathcal{D}_2)$ and $sf(T_1(u_1), T_2(u_2)) = \tau\prime \geq \tau$.

Of course, we can evaluate TSJ over the same database by specifying $\mathcal{D}_1 = \mathcal{D}_2$ or over a single XML tree collection by specifying $exp_1(\mathcal{D}_1)$=$exp_2(\mathcal{D}_2)$ or simply omitting the second parameter (i.e., as a self-similarity join).

Example 6.1. *Consider the XML database hospital.xml whose fragment is shown in Figure 6.2. Assume that an EDS query has returned the set of PCIs $\{3, 4\}$ (e.g., using the path specification /`patient`/`mother`/`name` and /`patient`/`name`). Then, we can identify all exams in hospital.xml whose similarity is not less than 0.75 using the following TSJ invocation:*

$$TSJ(doc(\text{``}hospital.xml\text{''})//exam, PCL_{tlc}, 0.75)\ ,$$

where PCL_{tlc} is the similarity function defined by the triple $\langle pcl[s, t, \{3, 4\}], IDF, wjs\rangle$.

6.2.2 A Glimpse on TSJ Evaluation

The course of the TSJ evaluation is shown in Figure 6.6. The following steps are executed:

1. *Tree Access*: In the first stage, the trees forming the input for TSJ are fetched to a main-memory resident area. To this end, services at L4 (e.g., the NodeMgr) are accessed twice: to select a list of DeweyIDs corresponding to the root nodes of the input trees (*rootID list*, for short) and, then, to fetch the corresponding trees.

2. *Profile Generation*: Using a tokenization function and the set PC_t, a profile is generated for each tree from 1. This step includes token annotation and character-level transformations on text data, namely, conversion to upper-case and elimination of repeated white spaces; further, all token values are hashed into integers. This step requires a PCR-PCI table as auxiliary information source. A PCR-PCI table is shown in Figure 6.3.

3. *Set Generation*: In this step, each profile is converted into a sorted set by sorting the tokens in increasing order of frequency in the data collection and, in case of using a weighting scheme, weights are calculated and attached to each token. This step requires a token-frequency table as auxiliary information source.

4. *Set Similarity Join*: After sorting the input sets in increasing order of the set size, a set similarity join is executed and pairs of root-node DeweyIDs—whose subtrees satisfy the similarity condition—are returned as result together with their similarity value.

The underlying similarity function of TSJ is determined by steps 2–4. As discussed earlier, we can easily obtain a rich repertoire of similarity functions by varying the combinations of these steps. This flexibility is a particularly appealing feature of our framework: we can freely plug-in different components or apply different parameterizations thereof to serve EM applications with various notions of similarity.

Note that steps 2 and 3 require two auxiliary information sources, the *PCR-PCI table* and the *token-frequency table*, respectively. The PCR-PCI table stores the association between path classes and path clusters. As we will show in Section 6.4.2, the generation of PCI-based profiles requires accessing the corresponding PCI for every node in a tree representation. By employing a separate hash-based map instead of "annotating" the PS as suggested in Chapter 3, we obtain much faster access times and avoid PS contention. The token-frequency table is necessary for sorting the profiles and for token weight calculation. Together with PCS, these structures represent the memory-resident structures needed for TSJ execution.

As suggested by Figure 6.6, we are not using any index scheme to provide access and maintenance on precomputed profiles (output of 2) or sorted sets (input of 4); all the computation is done on-the-fly. It is pertinent to discuss this aspect now.

Indexing techniques have mainly been used in the context of *similarity selection queries*, where the task is to approximately match an entity (string or XML tree) against a dataset [CGS03, ABG06, HCKS08, LLL08, CK09]—recall, we have conducted our accuracy experiments under this setting. Similarity selection solutions are typically designed for applications scenarios characterized by a large number of (concurrent) requests (for example, auto-completion applications where completion suggestions are obtained by approximately matching the partially entered string against the database [CK09]); hence, the cost of building an index structure is amortized across multiple uses. Moreover, the lookup time on the index is practically independent of the dataset size and orders of magnitude faster than building the index on-the-fly. Of course, the main issue is index maintenance in the presence of data updates. To avoid the burden of rebuilding the index from scratch every time the underlying data changes, arriving updates can be buffered and the index updated in batches, incrementally, at regular time intervals [ABG06, HKS09]. Unfortunately, index maintenance is particularly difficult when weighting schemes based on corpus statistics are employed (e.g., IDF). In such cases, even updates involving a single or a small set of entities can trigger a cascading effect, where weight updates have to propagated to many other entities.

As a result, supporting data structures are required and maintenance cost and space overhead are substantially increased.

Here, we focus on similarity joins in the EM context. Although we envisage the execution of our algorithms in operational databases serving read/write operations, we assume that concurrent TSJ invocations coming from multiple EM processes are quite rare. In contrast, we anticipate the common situation where a single EM process is carried out to identify duplicates at much more sparse intervals than those required by applications using similarity selection queries.[6] In this regard, multiple calls to the TSJ operator may occur using different parameters, in particular, thresholds, PC_t sets, and similarity functions. For different thresholds, we can simply materialize the input of step 4 (this approach is to some extent similar to creating *materialized views*); then subsequent TSJ executions only need to sequentially read stored data.[7] The use of multiple PC_t sets, tokenization methods, or both, generates different profiles for each tree. Thus, an index for each combination would have to be built and maintained. Furthermore, compared with similarity selection, similarity join is computationally much more intensive. As a result, profile and set generation do not take take an exceedingly large fraction of the overall processing time (e.g., see [CGK06] and our own results in Section 6.8). To perform on-the-fly evaluation, we only need to maintain a token-frequency table for set generation (step 3). As we will discuss in detail shortly, we can easily construct and, more importantly, easily update this table to reflect data changes. Finally, we note that on-the-fly evaluation is frequently the only option in several data integration scenarios, such as EII and dataspace systems. In the following, we focus on on-the-fly similarity join evaluation. In Section 6.7, we consider strategies for indexing and maintaining materialized sorted sets (input of step 4).

We will discuss steps 1 and 2 in the following two sections. Step 3 is straightforward while step 4 exactly corresponds to the set similarity joins extensively discussed in Chapter 5. We will then be ready to present the encapsulation of TSJ as a query evaluation plan (QEP, or simply plan) where each step corresponds to a physical operator (Section 6.6).

6.3 Tree Access

In this section, we first describe the exploitation of the index structures described in Section 6.1.4 to obtain the rootID list. Then, we discuss strategies for fetching the selected trees to a memory-resident area.

[6]Especially for static databases, e.g., data warehouses, a reasonable approach after the identification and elimination of duplicates is to avoid the appearance of new duplicates by employing similarity selection on all incoming data [CGS03].

[7]For this last scenario, we can employ the techniques described in [CGS03] to evaluate our algorithms progressively and avoid redundant computation of similarity results.

6.3.1 TSJ Input Selection

In principle, arbitrary XPath and XQuery expressions can be employed for selecting the TSJ input. We can therefore rely on the query optimization capabilities of XTC for selecting suitable indexes and deriving a cheap query plan. Moreover, such queries frequently require materialization of all subtrees rooted at the nodes contained in the result. Therefore, tree scan is normally already embedded in the evaluation of regular XML queries.

In Chapter 2, we have uncovered our assumptions about the underlying XML dataset: common vocabulary and tree-structured entity descriptions. The former assumption means that there is no source of semantic heterogeneity among node labels such synonyms and polysemes. The latter assumption ensures that the entities of interest are represented by trees with well-defined entry points, i.e., a common root-node label; this root-node label is used as target in the clustering process.

In the light of the assumptions above, sinple queries specifying the TSJ input have the form of "$//tgl$" or "$//path\_expression/tgl$", where *tgl* is the target label used for path clustering (see Chapter 3). The first query collects all subtrees rooted at *tgl* in an XML dataset and can be answered using the element index; the second query specifies a context using *path_expression* and can be answered using an element index which stores the PCR in the field value. Another scenario we have in mind is the matching of whole XML documents belonging to the same *collection*, i.e., sets of related documents. In XTC, collections are implemented by adding a virtual node under which all documents in a collection are appended. Thus, the query in this situation would be $/collection\_label/tgl$. In all cases, we have to remove DeweyIDs related to nested occurrences of *tgl*. This can be easily done by scanning the list of DeweyIDs and by removing every DeweyID that is contained in the previous DeweyID. Note that the target label may appear in different contexts (i.e., hierarchical nesting) or collections. Hence, for binary joins, the use of different expressions for locating the target label may be necessary.

Further, one can specify predicate constraints in the query for selecting a subset of the trees rooted at tgl. The descendant axis and wildcards are often used in the predicate to overcome structural heterogeneity. A example of such a query is "$//tgl[232 < //study/id < 282]$", which selects the trees having the *study* identifier within the specified range. For this query, a CAS index would be beneficial. If a suitable CAS index and an element index are available, a typical query evaluation plan would consist of using the element index to obtain a list of DeweyIDs relative to tgl (L_{tlg}) and the CAS index to obtain a list of DeweyIDs relative to content nodes satisfying the predicate constraint (L_{pred}). If the CAS index was defined with a more general path query than that specified in the predicate (e.g., $//id$), then L_{pred} may contain some false positives that have to be filtered out (see [Mat09] for details about how false positives are identified). Then, a structural join [AKJP+02] using the ancestor relationship is performed between L_{tgl} and L_{pred} and a duplicate-free subset of L_{tgl} is returned.

We now illustrate how to exploit the PCS structure (Chapter 4) and the flexible indexing model of XTC to embody contextual information in the predicate constraint while maintaining the resilience against structural heterogeneity. For this purpose, we just need to embed an EDS query (see Definition 4.6) into the tree selection expression. For example, a query of the form "$//tgl[value1 < eds(eds\_path)/value() < value2]$"[8] could be specified. This expression will return sub-trees such that: 1) the root-node label is *tgl*; 2) contains a least one path that *approximately* matches *eds_path*, i.e., a path that is associated with a PCI returned by the *eds* function; 3) the content value under the matched path lies within the range $[value1, value2]$. Note that PCS provides more flexibility regarding approximate path matching than employing descendant axis and wildcards. For example, given $eds\_path = a/b$, we can not only match content nodes under paths such as a/b or $a//b$, but also paths like b/a; this latter nesting variation cannot be easily captured using XPath expressions.

Under the hood, the query compiler translates a call to eds in the path specification into a query on the PCS structure and the returned PCIs are automatically inserted in the query evaluation.[9] Now, we can benefit from the flexibility of CAS indexes: instead of PCRs, we can store PCIs in the CAS index record. The only modification needed for building and maintaining such CAS indexes is accessing the PCR-PCI table to obtain the PCI associated with each PCR. Likewise, the query answering process is practically the same as for ordinary path predicates: the CAS index is probed to obtain the list L_{pred}, false positives (in this case, DeweyIDs that are not associated with a PCI returned by PCS) are removed from L_{pred}; a structural join between L_{tgl} and L_{pred} takes places; and, finally, a subset of L_{tgl} is returned.

Note that the purpose of the EDS query here is totally different from that presented in Chapter 4. There, EDS queries are employed to define the portion of textual content used to describe entities, whereas, here, they support filtering of the similarity join input. This observation highlights the flexibility of the PCS structure—and the path clustering approach in general, which can provide approximate path matching functionality for arbitrary applications. Also note that we can only use EDS queries for filtering the input; approximately selecting the initial set of trees is not possible because tree selection is defined on the same label that was used as entry point for path clustering, i.e., the target label. Hence, similarity information about partial paths leading from the root document node to the target label, which would be necessary for enabling approximate tree selection, is not covered by PCS structure.

6.3.2 TSJ Input Scan

In Section 6.1.3, we have sketched the tree scan algorithm. Actually, XTC provides the method *getScanPartition* at L4, which allows to scan arbitrary portions of a document or collection. This method receives two DeweyIDs as parameters that determine the start and end points for the scan process. To fetch a single subtree, a caller has to

[8] Other parameters of the EDS query are omitted for simplicity.

[9] As already mentioned, we have not integrated our framework into XTC's query optimizer yet; currently, this functionality is hardwired in the access module at L4.

pass the same root-node DeweyID to both parameters. Additionally, *getScanPartition* can take a set of predicates as arguments; these predicates are applied on every node scanned (or reconstructed if virtual) and only nodes satisfying the predicates are returned—such predicates are commonly referred to as SARGS (simple search arguments) in the relational world. An important kind of predicate in our context tests the node type to qualify only element nodes or text nodes. The latter case enables an optimized scan process in path-oriented storage model, because inner-node reconstruction can be bypassed.

The *geScanPartition* method is encapsulated in a physical operator in XTC, where a complete ONC iteration corresponds to a scan process (from the start point to the end point); in our case, it coincides to a tree scan. To avoid excessive function calls to the *next* method and the overhead of a per-node processing strategy, we have reimplemented *geScanPartion* following a block-oriented iterator model [PMAJ01]. In this model, an ONC iteration performs a single call to *next*, in which the whole tree is returned as a block to the consumer operator.

Given the list of DeweyIDs returned by a tree selection expression, the corresponding trees can be fetched by invoking the *geScanPartition* partition operator for each DeweyID. Every *getScanPartition* execution traverses the document index to locate the page where the first tree node is contained, and then proceeds sequentially along the chained pages until the last node is found. Because the list of DeweyIDs is typically large, the cost of many index traversals can be excessive. Moreover, several data pages are likely to be touched multiple times. As for relational systems, an alternative to an index scan is to perform a full scan over the whole XML database to collect the trees of interest. The main advantage of this strategy is sequential I/O. On the flip side, this process touches all container pages even when all input trees are clustered within a relatively small portion of the database store.

Disregarding the distribution of the trees in the database, we can consider the tree selectivity to roughly estimate the scan performance. In our experiments in XTC, we achieved better performance with *geScanPartition* even when 50% of the XML trees were accessed. In the following, we will only consider the *geScanPartition* operator. The definition of a suitable cost model is a future task.

6.4 Profile Generation

We now present the algorithms used in the profile generation operators. For ordered trees, we describe the generation of the *epq*[v2] profile; for unordered trees, we describe the generation of the *pcl*[t] profile. Of course, *any* tokenization method on tree-structured data can be used in our framework (all tokenization functions discussed in this thesis are currently supported). As *getScanPartition*, each operator is implemented as a block-oriented operator: they receive a set of nodes representing an XML tree as input and output a set of tokens. Both algorithms generate tokens containing textual information based on the set of PCIs returned by an EDS query, i.e., the PC_t set. We define the tokenization function *qgram* with an extra Boolean pa-

Algorithm 6.1: Algorithm for the generation of *epq*-gram tokens

```
Input: A tree $T$, positive integers $p$ and $q$, the set $PC_t$
Output: The $epq[p, q, v2, PCR_t](T)$ profile (abbreviated to $\mathcal{P}$)
1  ...
13 foreach for each child c (from left to right) from u do
14     if c is a text node then
15         if $c.pcr \in PCR_t$ then
16             $qnull \leftarrow$ concatenation of q $*$
17             $anc\text{-}p \leftarrow shift(anc, *)$
18             foreach $tok \in qgram(c, q, false)$ do
19                 $sib \leftarrow shift(sib, tok)$
20                 $\mathcal{P} \leftarrow \mathcal{P} \cup (anc \circ sib)$
21                 $anc\text{-}p \leftarrow tail(anc\text{-}p, tok)$
22                 $\mathcal{P} \leftarrow \mathcal{P} \cup (anc\text{-}p \circ qnull)$
23         continue
24     ...
25 ...
26 return $I$
```

rameter specifying whether the text node string is extended with null symbols or not. This function also embodies preprocessing operations, i.e., conversion to upper-case and elimination of repeated white spaces.

6.4.1 Ordered Trees

The algorithm for the generation of *epq-v2* tokens we implemented is very close to the algorithm for the generation of *pq*-grams presented by Augsten et al. [ABG10]. The difference, of course, lies in the way we handle text nodes. Thus, we only discuss this aspect of the algorithm here. We follow the same notation of the original algorithm. The stem and the base are represented by two shift registers: *anc* of size *p* and sib of size *q*. The *shift* operation is used to remove the head of the queue and insert a new element at the tail of the queue. For example, if the content of *anc* is (*a*,*b*,*c*), then we have $shift(anc, x) = (b, c, x)$. For ease of presentation, we define the additional operation *tail*, which substitutes the element at the tail of the queue by a new element, i.e., $tail((a, b, c), x) = (a, b, x)$. As for tokens, the concatenation of the two registers is denoted by $anc \circ sib$. In the following, we will make no distinction between nodes and *q*-gram tokens.

A snippet of the algorithm for the generation of *epq*[v2] profiles is listed in Algorithm 6.1. The algorithm has parameter PCR_t, which corresponds to the set of PCRs associated with the elements of PC_t. This set is constructed before the first execution of the algorithm by doing a sweep on the list of key/value pairs of the PCR-PCI table. As a consequence, we avoid accessing the PCR-PCI table many times for each input

Algorithm 6.2: The algorithm for generation of PCI-based tokens

```
Input: A set of text nodes and null nodes $N$ from a tree $T$, an positive integer $q$, the set
       $PC_t$, the PCR-PCI table
Output: The $pci_{tlc}[q, PC_t](T)$ profile (abbreviated to $\mathcal{P}$)
1 foreach $u \in N$ do
2     $pci \leftarrow PCR\text{-}PCI(u.pcr)$
3     if $u \in PC_t$ then
4         foreach $tok \in qgram(u, q, true)$ do
5             $\mathcal{P} \leftarrow \mathcal{P} \cup (pci \circ tok)$  // insert new pci-qgram into the
                  profile
6     else
7         $\mathcal{P} \leftarrow \mathcal{P} \cup pci$  // insert new pci into the profile
8 return $\mathcal{P}$
```

tree. When iterating over the children of a node u, we first check if the current child is a text node (line 14). If it is the case, we verify if its corresponding PCR is in the PCR_t set (line 15). If this is case, we generate the corresponding *epq*-v2 tokens (lines 16–22); otherwise the node is ignored and the algorithm proceeds to the next child (line 23). epq-v2 tokens are formed by stems having either u as anchor (represented by *anc*) or a *q*-gram token (represented by the register *anc-p*). The base of the epq-v2 tokens are either appended by a token (represented by *sib*) or by a concatenation of q $*$ symbols (represented by *qnull*). In the loop at line 18, the algorithm iterates over the set of tokens returned by the *qgram* function and composes *epq*-v2 tokens by concatenating *anc* with *sib* and *anc-p* with *qnull*. Note that we do not extend the text node value with null symbols—we call the *qgram* function passing *false* as parameter; the utility of null symbols (see Chapter 2, Section 2.3.2) is already provided by the dummy nodes (represented by the $*$ symbol).

6.4.2 Unordered Trees

The algorithm for the generation of *PCI*-based profiles is shown in Algorithm 6.2. The simplicity of the algorithm reflects the convenience of the path-oriented storage model to our PCI-based tokenization method. First, only text nodes and null nodes (recall that null nodes represent paths that are not associated to a text node) are needed to derive a tree representation. Hence, the reconstruction of inner nodes by the *getScanPartition* operator is obviated. Further, PCIs are obtained by a simple lookup at the PCR-PCI table (line 2); the corresponding tokens are obtained directly from the PCI value (line 6) or by the concatenation $pci \circ tok$ (line 4). As a result, tree scan and profile generation are implemented by lightweight operators that impose very little overhead to the overall similarity processing. This fact strengthens the case for on-the-fly evaluation that we argued previously.

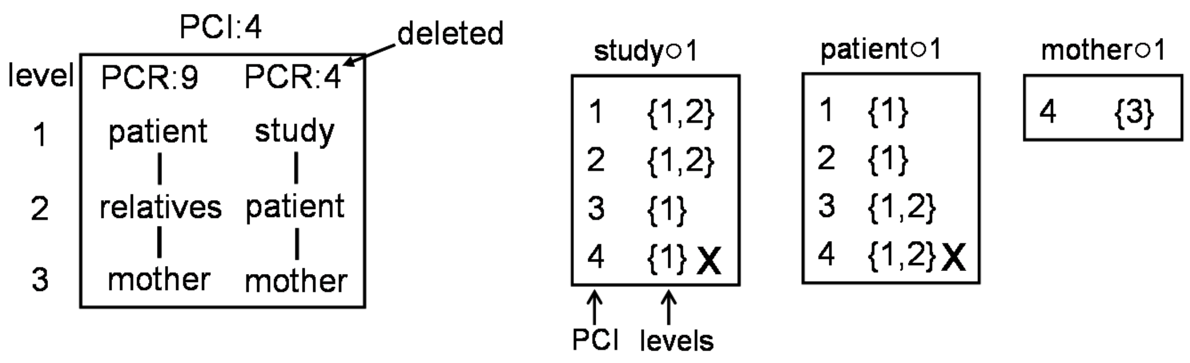


Figure 6.7: A path cluster and a sample of the PCS represented as inverted lists

6.5 Auxiliary Structures: Building and Maintenance

Our framework has to maintain three auxiliary data structures: the PCS, the PCR-PCI table, and the token-frequency table. All of them are kept memory-resident during similarity join evaluation and are incrementally updated as the database state changes. We identify that a node is covered by the clustering process by checking if the path from the root to the node (including the node itself) contains at least one element whose label is the same as the target label. At the time of writing, the update functionality of our framework is not fully implemented in XTC yet. Nevertheless, we already provide here a detailed discussion on the engineering issues around the implementation of this feature. In the following, we first present the general approach for building and maintaining these auxiliary data structures; afterwards, we discuss how we can reduce the maintenance cost.

6.5.1 PCS and PCR-PCI Table

The PCS and the PCR-PCI table are built at the end of the path clustering process. Both structures have a reasonably small memory footprint: the size of the PCS was already discussed in Chapter 4; the PCR-PCI table requires 4 bytes per entry (two bytes each for PCR and PCI) where the number of entries corresponds to the number of distinct paths in the dataset. Modifications on the PS, i.e., insertion or deletion of path classes, have to be propagated to these structures. For the PCR-PCI table, this task is trivial: we only need to remove or insert a table entry. However, the PCS maintenance demands more explanation.

Let us first explain the handling of deletions using an example. Consider the deletion the path class `/hospital/exam/study/patient/mother`, whose PCR is 13 and the associated PCI is 4 (see Figure 6.3). We remove the entry $\langle 13, 4 \rangle$ from the PCR-PCI table but keep the PCI value. We then access the catalog to obtain the target label used in the clustering process (i.e., `exam`), remove from the path all labels from the root up to the target label (i.e., `hospital` and `exam`) and derive the path profile $\{\texttt{study} \circ 1, \texttt{patient} \circ 1, \texttt{mother} \circ 1\}$, which is used to access the corresponding inverted lists in the PCS. For convenience, the inverted lists from Figure 4.3(b) on Page 96 are repeated in Figure 6.7 together with the path cluster whose PCI is 4. For the token $\texttt{study} \circ 1$, we scan the associated inverted list until we find the entry $\langle \texttt{PCI} : 4, \texttt{levels} : \{1\}\rangle$. Because $\texttt{study} \circ 1$ appears (starting from the target label) at the first level in the deleted path class, we remove this entry from the inverted list. Proceeding similarly, we scan the inverted list associated with the token $\texttt{patient} \circ 1$ and find the entry $\langle \texttt{PCI} : 4, \texttt{levels} : \{1, 2\}\rangle$; then we remove the second value of the field `levels`, i.e., 2. The last inverted list is associated with the token $\texttt{mother} \circ 1$, which contains the record $\langle \texttt{PCI} : 4, \texttt{levels} : \{3\}\rangle$. However, now we cannot remove this record, because the cluster prototype contains another path class in which the label `mother` appears at level 3. To know whether we can update or remove a record from an inverted list, we need the information about how many path classes "contain" the corresponding token at a given level. To this end, we store this information together with each each level value in the inverted lists. The new entry layout is $\langle \texttt{PCI}, \texttt{levels} : |\texttt{PCR}|\rangle$, which requires two bytes for each level information. In the inverted list associated with the $\texttt{mother} \circ 1$ token, the entry is $\langle \texttt{PCI} : 4, \texttt{levels} : \{3 : 2\}\rangle$. Hence, instead of deleting the entry, we update the $|PCR|$ value from 2 to 1.

As already described in Chapter 4, insertion of a new path class triggers a match of this path class against the PCS. If no cluster prototype is returned having a similarity to the new path class which is not less than a specified threshold, a new cluster is created and the path class is assigned to it. Otherwise, we assign the path class to the most similar cluster prototype. In both cases, we have to add a new entry to the PCR-PCI table and perform appropriate operations to update the PCS, i.e., update of the $|PCR|$ information, insertion of level information or entries, or creation of new inverted lists.

6.5.2 Token-Frequency Table

We build the token-frequency table by performing a single sweep over the database, typically right after the clustering process. During the scan, we also collect path cluster statistics, in particular, statistics about the string length of text nodes appearing in the cluster, which are stored in the catalog.

As distinct PC_t sets yield different token sets, we need a mechanism to provide token-frequency information for arbitrary EDS queries. In principle, we adopt the simple solution of generating all possible tokens, i.e., we generate tokens for $PC = PC_t$ (i.e., $PC_s = \oslash$) and $PC = PC_s$ (i.e., $PC_t = \oslash$). For *PCI*-based profiles, we use a slightly adapted version of Algorithm 6.2, where lines 4–5 and line 7 are executed for all input nodes. For *epq*-gram profiles, we execute both the Algorithm 6.1 and the original *pq*-gram algorithm and take the duplicate-free union of the resulting profiles. Note that we have to build a token-frequency table for each tokenization method as well as for different parameterizations thereof (e.g., *q*-gram size).

Despite the large number of tokens, the frequency table is still small enough to typically fit into main memory. For example, using Nasa and SwissProt, we generated 37 K and 19 K distinct tokens, respectively (TLC similarity function, *q*-gram of size 2). The frequency table requires 8 bytes per entry (four bytes each for the hashed token and its frequency); thus, only 36KB are sufficient to keep the frequencies of all Nasa tokens memory-resident, while SwissProt needs 19 KB. In rare cases when the available memory is insufficient, we can restrict the size of the token-frequency table to at most *E* entries. In this approach, multiple tokens will collapse into a single entry and their frequencies are summed up. Such collisions negatively affect accuracy—because incorrect token weights are computed—and efficiency—because rare tokens may be shifted away from the prefix positions due to their incorrectly increased frequency value. We have not measured the impact of size-constrained tables on our algorithms. In our experiments, we assume that the token-frequency table entirely fits into main memory.

Note that we already admit some inaccuracy in the similarity results by hashing token values. In case of collisions, distinct tokens receive the same hash value. Such collisions not only incorrectly increase token-frequency values but also set-overlap results, because different tokens are matched. Fortunately, collisions can be neglected in practice. Using the Karp-Rabin fingerprint function [KR87], we measured less than 0.01% of collisions in all datasets. More importantly, we have not observed any significant degradation of the accuracy results. Besides reducing the size of the token-frequency table, token representation as integer values significantly improves the overall efficiency of similarity join processing.

For *PCI*-based tokens, updating the token-frequency table after data changes is easy. In case of deleting structure nodes, content nodes, or both, we only need to generate the tokens for the deleted data to probe the token-frequency table and decrease the corresponding frequency by one—tokens with frequency zero are removed from the table; in case of insertions, we generate the tokens for the new data and increment their frequency accordingly or add an entry in the token-frequency table for new tokens. For *epq*-gram tokens, incremental updates are more complicated due to the underlying sibling ordering which imposes more data dependency on the token generation process. We have not yet investigated whether or not the techniques presented in [ABG06] for incremental updates of *pq*-gram profiles can be adapted for *epq*-grams. Hence, we currently apply the profile generation algorithm on the whole tree—on the old and the new version—to update the token-frequency table.

6.5.3 Reducing Maintenance Cost

We now discuss ways to improve maintenance efficiency and reduce storage space of the auxiliary data structures. The first approach consists of adopting lazy update propagation [ABG06, HKS09]. As already mentioned, using this scheme, updates are buffered and propagated incrementally to similarity indexes at fixed time intervals. We can straightforwardly implement this strategy in our framework. Of course, the inconvenience of this approach is that inaccurate results are reported when similarity operations are invoked between updates. We also need extra space to store the log of data modifications.

The next strategy both saves storage space and improves performance. We have already argued that the choice of the textual elements that will compose the content part is driven by their semantic properties whose value is dependent on the underlying application, domain, and data characteristics. In our framework, we provide the user with extensive flexibility in specifying such textual elements. Nevertheless, there exist commonplace assumptions that allow us to make this task semi-automatic. First, long strings are typically deemed as inappropriate for EM, because their "semantic ambit" is too large and there is too much leeway for deviations. Short strings are preferable because they are often used to name or concisely describe essential information about an entity. Second, text nodes appearing under infrequent paths correspond to fields exhibiting high incidence of NULL values in the relational context. While there are many conceivable interpretations for the absence of explicit values, the fact that a field is commonly left unspecified suggests that this field is unsuitable for distinguishing entities from one another in the context of the entire database—of course, it might still be useful among those entities in which the field value is given. Moreover, for *PCI*-based profiles, tokens containing textual information inherit the frequency of the corresponding path cluster, i.e., their frequency in the data collection is less than or equal to that of its path cluster. Hence, tokens derived from rare PCIs are assigned very high IDF weights. As a result, entities represented by profiles containing such tokens are likely to always yield low similarity results when compared with any other entity whose profile does not contain these tokens.

Given the considerations above, an intuitive strategy is to remove from the PCS those path clusters that exhibit low-frequency or contain paths leading to long strings. By not representing these path clusters in the PCS, we prevent them from being used to compose the textual representation of entities, i.e., such path clusters are fixed in PC_s—henceforth, these path clusters are denoted by PC_{s*}. Thus, the size of the PCS and the token-frequency table are substantially reduced. The savings are particularly dramatic for the token-frequency table, because long strings are responsible for a large portion of distinct tokens. Moreover, path classes with low-support in the database are exactly those that cause the bulk of updates in the PS and, consequently, in the PCS. Thus, PCS maintenance workload is also greatly reduced.

The set of path clusters to be removed from the PCS is identified during the building phase of the token-frequency table. For each token, we store in a temporary table its PCI and a flag signaling whether or not it contains textual information. As before, we also collect path cluster statistics; at the end of the scan process, this information is used to select the set to be removed. As we create the token-frequency table from the temporary table, we leave out all textual tokens associated with a PCI in PC_{s*}. Finally, we remove from the PCS all entries related to elements PC_{s*}. Note that we do not remove entries associated with PCIs in PC_{s*} from the PCR-PCI table as they are still necessary to produce (*PCI*-based) structural tokens. Finally, we use the PCR-PCI table to identify whether or not a PCR is related to a path cluster in PC_{s*}. For this task, we use the most significant bit in the two-byte PCI representation to indicate whether or not a PCI is an element of PC_{s*}[10].

Parameters used for determining the elements of PC_{s*} are the percentage of trees in which any element of a path cluster appears and the percentage of strings with size larger than a given value. For example, we can assign to PC_{s*} path clusters that appear in less than 75% of the trees in the collection or have more than 3% of text nodes with size larger than 100. In this regard, lazy update propagation is appropriate for PCS maintenance. Periodically, we can check for path clusters whose frequency has (decreased) increased (below) above the specified threshold and update the PCS accordingly. Currently, we do not keep track of the percentage of long strings in a path cluster after having built the token-frequency table; supporting this feature is part of future work.

6.6 TSJ as a Query Evaluation Plan

We now put all TSJ components together to compose a complete similarity join operator. To this end, the main design objectives are *seamless integration* of the TSJ operator into XTC's architecture and *performance*. The former objective is achieved by assembling all TSJ components into a QEP, which leverages existing physical operators for regular XML query processing while the performance objective is obtained by enabling pipelining as much as possible. Currently, TSJ is executed directly at L4 as a "pre-canned" transaction using statically defined access paths and is parameterized using TSJ-specific variables. We only describe the evaluation of TSJ here. Preprocessing steps, i.e., path clustering, building of auxiliary data structures, and EDS query answering have already been covered earlier in this chapter and in Chapter 4.

Figure 6.8 illustrates an example for a TSJ plan. Indexes and auxiliary structures are also shown in the figure. Physical operators are represented by rounded rectangles; operators specifically designed for similarity join processing are highlighted with a shade of gray. For ease of exposition, the XML processing operators have simpler names than those defined in the original specification [Mat09]. Parts of the operator tree that closely correspond to the steps in the abstract course of TSJ evaluation (see Figure 6.6) are identified with the respective numbers.

At a glance, the branch at the lower end of the operator tree executes input selection (left-hand side) and fetching and conversion of trees into sorted sets (right-hand side). All lowest-level operators communicate with the access services layer (L3) through the NodeMgr. The fork of the branch directs DeweyIDs from the left-hand side to right-hand side and, later, combines these DeweyIDs with the resulting sorted sets coming from the right-hand side in a tuple, and delivers it upwards in the tree. Lastly, the stem at the higher end receives these tuples and outputs pairs of DeweyIDs whose corresponding XML trees satisfy the similarity predicate together with the corresponding similarity score.

[10]As PCIs represent groups of path classes, it is reasonable to represent them using half of the range available for PCRs.

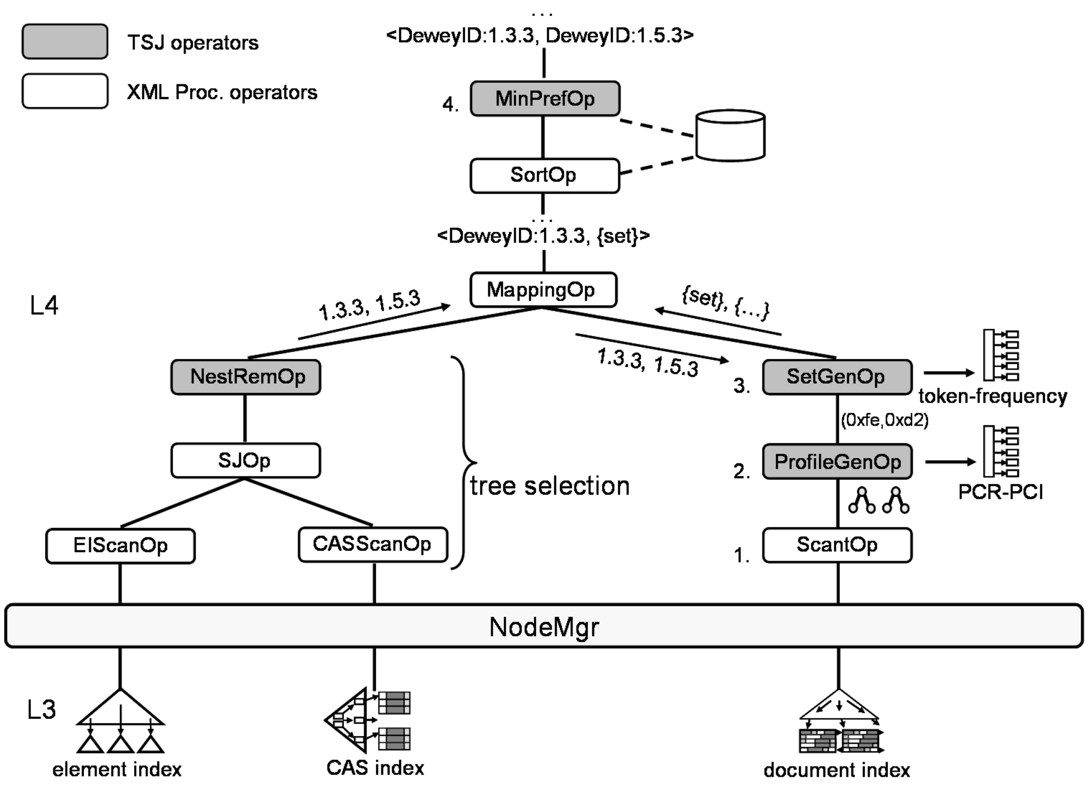


Figure 6.8: TSJ query evaluation plan

In more detail, and following the dataflow order, the tree input selection strategy used in this QEP exploits the element index for obtaining an initial list of root-node DeweyIDs, which are filtered by predicates evaluated using a CAS index. Access to these indexes is implemented by the operators *ElScanOp* and *CASScanOp*. The structural join operator (*SJOp*) is used to identify the subset of the DeweyIDs delivered by *ElScanOp* that are ancestors of a least one DeweyID in the list returned by *CASScanOp*. The next operator, *NestRemOp*, simply removes nested DeweyID sequences by retaining only the topmost DeweyID. Completing the Tree Access step, the list of DeweyIDs is streamed along the path to the *getScanPartition* operator (abbreviated to *ScanOp* in the figure), which fetches a tree at a time using the document index. The following two components upwards are straightforward implementations of the steps 2 (*Profile Generation*) and 3 (*Set Generation*): trees represented by sets of nodes are converted into profiles by *ProfileGenOp* with the support of the PCR-PCI table and, afterwards, the profiles are converted to sorted sets with the support of the token-frequency table. Tuples containing a root DeweyID and the corresponding sorted set are constructed by *MappingOp* and sorted in increasing order of the set size by *SortOp*. Alternatively, the output of *SortOp* can be saved on disk to avoid repeated execution of steps 1–3. Finally, the *Set Similarity* step is performed by *MinPrefOp* and scored pairs of DeweyIDs are delivered to the TSJ consumer.

Note that TSJ can not only be used as stand-alone operator, but also as part of more complex XML queries. For example, we can simply plug-in a sort operator on top of the operator tree for delivering the resulting DeweyIDs in document order.

6.7 Further Integration

In this section, we discuss future work on providing tighter integration of our framework into XTC. We focus on the issues around making TSJ a first-class database operator; most of them represent open research problems. We also cover the support of EDS queries as part of the TSJ evaluation, i.e., when EDS queries are embedded into the similarity join as a sub-query or used for approximate filtering of the TSJ input (automatic EDS queries with *stats* parameter set to *False*). Preprocessing steps such as evaluation of interactive and exploratory EDS queries and path clustering are not discussed. The former can be performed by combining automatic EDS queries and regular queries on PCS statistics. For this purpose, we can define a schema for the PCS statistics stored in the catalog and provide predefined views over it. Path clustering can be triggered by a DDL statement for creating the PCS structure.

Specifically, we discuss the following three aspects in this section: extensions to existing query languages, query processing, and materialized input maintenance. Along the discussion, we pinpoint relevant work in the literature which can be leveraged to support our endeavor.

6.7.1 Query Language

To express TSJ queries using XML languages, we need (implicit or explicit) constructs for EDS queries and the TSJ operation itself. In general, a source language providing syntactical elements and clean semantics is needed to express operations under the similarity matching paradigm. The two most popular XML query languages supporting similarity matching are *W3C XQuery and XPath Full Text 1.0* (*XQuery Full-Text*, for short) ([AYBB+10, AYBS04] and *Narrowed Extended XPath I* (NEXI) ([TS04]. XQuery Full-Text extends XQuery and XPath languages with full-text search primitives, such as word search, scoring, and top-K ranking. NEXI is the official language of the *Initiative for the Evaluation of XML Retrieval* (INEX) [FGKL02], which is a campaign for evaluating XML search systems under common test data collections and effectiveness measurements paradigms. Although being designed for XML search, both languages present features and underlying concepts that can be reused or extended to express TSJ queries.

XQuery Full-Text extends XQuery with a full-text search expression called *FTContainsExpr*, which receives a *search context expression* and a search condition called *FT-Selection*. *FTContainsExpr* behaves like a regular XQuery comparison expression (i.e., expression involving operators such as *eq*, $<$, and $<=$) by returning a Boolean in the XQuery data model—it returns $True$, if some node in the sequence of XML nodes selected by the search context expression satisfies the search condition, and $False$, otherwise.

More relevant to our context are scoring functions and the support of matching modifiers. Let us discuss first the scoring functions. The functions *fts:scoreSequence()* and *fts:score()* associate a numeric score (in the interval $[0, 1]$) with the result of XQuery expressions; the underlying scoring mechanism is implementation-dependent. TSJ already adheres to the semantics of *ft:score()*, because each pair of nodes in the join result is associated with its similarity score. Hence, to express a TSJ query, we can pass on a regular join expression with join partners defined by search context expressions (or tree selection expression in our nomenclature) to *ft:score()*, where the scoring algorithm is defined by the similarity function. In this regard, the expression below is similar to the TSJ invocation presented in Example 6.1.

```
for $r score $s in
    (for $a in \funca{doc}{``hospital.xml''},
         $b in \funca{doc}{``hospital.xml''}
    where $a//exam eq $b//exam
    return <match>{$a}{$b}</match>)
where $s >= 0.75
return <result>{$r,<score>$s</score>}</result>
```

In the XQuery expression, the score variable added to the *for* clause of the FLWOR expression is syntactic sugar. When such an expression is compiled, the inner *for* expression is passed on as parameter on to the *fts:scoreSequence()*[11]. Although not stated in the XQuery Full-Text specification, similarity matching is a reasonable semantics for the value comparison operator *eq* on XML nodes when scores are evaluated. Unfortunately, at the time of writing, XQuery Full-Text [AYBB+10] does not support a truly flexible scoring construct where different scoring algorithms (in our case, similarity functions) can be specified and passed as parameter on to the scoring function.

A rich repertoire of matching modifiers, called *FTMatchOptions*, is available in XQuery Full-Text including stemming, stop words, case sensitivity, diacritics, special characters, and synonyms. Such modifiers can be used to specify data transformation operations on the content part of the TSJ input trees. Currently, *FTMatchOptions* can only be specified in the context of an *FTContainsExpr*. To use *FTMatchOptions* in the TSJ query above, it would be necessary to allow matching modifiers with regular XQuery expressions in the presence of score variables.

NEXI is a derivative of XPath with several simplifications and some extensions that focus on expressing XML search queries based on keywords (content-only queries) and keywords together with structural constraints (content-and-structure queries). In the latter query type, structural constraints can be interpreted as strict or vague (i.e., structural constraints are considered query hints). Query evaluation following the vague interpretation implies some sort of approximate path matching and, therefore,

[11]Hence, scoring functions are second-order functions as they receive an expression as parameter instead of a sequence of items.

resembles EDS queries. In XQuery Full-Text, on the other hand, there is no explicit support for loose interpretation of structural constraints; it has to be provided by specific implementations of the scoring mechanism.

In summary, XQuery Full-Text is the preferable language for supporting TSJ as it is fully composable with XQuery and XPath. To entirely support TSJ features, it would be necessary to extend XQuery Full-Text with the following functionalities: flexible specification of scoring algorithms, support of matching modifiers with ordinary XQuery expressions (at least when scores are evaluated), and an explicit construct dictating the interpretation of structural constraints (either strict or vague). We make no claims that these extensions can be easily incorporated into XQuery Full-Text, though.

6.7.2 Query Processing

For query processing, TSJ and EDS expressions in the source language must be detected by the query compiler and transformed to a QEP as the one depicted in Figure 6.8. Along this course, these operators have to cross some abstraction layers under different representations. In order to discuss aspects of the integration of our similarity operators in the query engine, we first have to provide an overview on the query compilation process of XTC.

XTC's Query Engine

In a nutshell, the following steps are performed to generate a QEP for an XQuery expression [Mat09]:

1. *Parsing*: The parser reads an XQuery expression represented as a string, analyzes its syntax, and, if no syntactic error is detected, outputs an *abstract syntax tree* (AST) describing the grammatical structure of the query expression.

2. *Translation*: The AST representation is transformed into XTC's logical query representation called *XML Query Graph Model* (XQGM). This model entails a graphical representation to which a logical algebra is attached. The AST-to-XQGM transformation is carried out by the *query translator*, which performs the following four operations in this order: *normalization*, which transforms the original query to its core representation (e.g., any syntactic sugar is removed); *static typing*, which infers (some) returning types of XQuery expressions; simplification, which removes superfluous subexpressions; *XQGM transformation*, which generates the XQGM representation by recursively matching patterns in the AST representation to *transformation rules*.

3. *Query Rewriting*: Several algebraic laws are used to produce alternative logical representations of the query expression. This operation is called query rewriting and is performed using a set of rewriting rules that transform an XQGM element into another one (and, hence, are called *XQGM-to-XQGM* transformations). In

particular, the initial logical plan produced by the AST-XQGM transformation closely follows the XQuery Formal Semantics specification and is characterized by typically inefficient node-at-a-time navigational operations and nested subexpressions. Hence, besides classical *algebraic rewriting* such as *selection pushdown*, query unnesting to enable set-at-a-time processing plays an important role at this compilation stage.

4. *Plan Generation*: Several alternative QEPs are produced using a set of *XQGM-to-Plan transformations*—the relationship between logical and physical operators expressed by XQGM-to-Plan transformations are not necessarily one-to-one: other cardinalities, namely, *1:n*, *n:1*, and *n:m* are also possible—and the *cheapest* QEP is selected for executing the query[12].

In XTC, the search space for optimization, i.e., the plan space derived from the set of XQGM-to-XQGM and XQGM-to-Plan transformations is generated in an interleaved fashion: as the rule engine traverses the initial XQGM tree, every logical rule match leading to an XQGM-to-XQGM transformation triggers the corresponding XQGM-to-Plan rule, where a description of the physical operators is created and attached to generated XQGM operator; at the end of the process, the XQGM tree is traversed again, and different QEP alternatives are built according to the attached physical operator descriptions. Note that XQGM is used for both logical and physical query representations. As cost-based optimization of XTC is not mature yet, rule-based transformations are performed and QEP selection is done heuristically.

TSJ Integration and Optimization Opportunities

A straightforward approach to integrate the TSJ and EDS queries into the query engine would be to implement them as XQuery functions. For example, we can name similarity functions and declare them as static variables and "unnest" the tokenizer parameter PC_t, i.e., define it as a parameter of the TSJ function (a sequence of atomic numeric values); similarly, we can support EDS queries by defining a function, say eds, which receives as parameter a sequence of strings denoting query paths. Further, as functions are handled by a common rewriting rule in XTC, we only need to define XQGM-to-Plan transformation rules for each TSJ instance (defined by the named similarity function) and for the eds function. Of course, many other details have to be addressed to realize this approach, such as exploitation of materialized input and index eligibility when evaluating tree selection expressions involving EDS queries[13], which will require further transformation rules with complex conditions of applicability. Fortunately, given the query engine design of XTC and its extensibility support, we see neither conceptual nor technical obstacles.

[12] In the approach adopted in XTC, a QEP is not a description of a program that can be understood by an interpreter or compiled, but it is the program itself (i.e., directly executable code is associated with the XQGM construct).

[13] In such cases, when no CAS index is available, we use a SARGS predicate—passed as parameter to the *getScanPartition* method—for filtering input trees based on the EDS query result.

In the above approach, similarity operations are handled as functions at the algebraic level and QEPs are constructed based solely on conditions of applicability. However, we believe that TSJ plan generation can be substantially improved by explicit algebraic treatment and, in particular, cost-based decisions as we motivate in the remainder of this section.

In [SAA10], Silva et al. present algebraic rules concerning several similarity operators. We adapt two of these rules to our context here. In this paper, these rules are defined on *range distance-join* operators over relational tables, which is the counterpart of a similarity join when a distance function is employed. Nonetheless, owing to the intrinsic duality between distance and similarity functions, the equivalences remain valid for TSJ. Further, we also represent the algebraic rules on XQuery expressions instead of tables. Regarding the notation, $E[p : E']$ denotes an expression E with a filtering predicate involving the output of expression E'; and $E_1 \overset{\approx}{\bowtie} E_2$ denotes a TSJ with E_1 and E_2 as tree selection expressions.

a.1: $E[p : E_1](E_1 \overset{\approx}{\bowtie} E_2) \equiv E(E_1[p] \overset{\approx}{\bowtie} E_2)$

a.2: $E[p : E_1](E_1[p_1] \overset{\approx}{\bowtie} E_2) \equiv E(E_1[p \wedge p_1] \overset{\approx}{\bowtie} E_2)$

b: $E_1 \overset{\approx}{\bowtie} E_2 \equiv E_2 \overset{\approx}{\bowtie} E_1$

Rules $a.1$ and $a.2$ ensure the applicability of the well-known *predicate push-down* technique. This means that predicates filtering the output of TSJ can be pushed to the tree selection expression. As similarity operations are in most cases much more expensive than filter predicates, employing this technique as a rewriting rule is likely to yield less expensive plans. In XTC, the XQGM rule engine analyzes selection predicates containing reference variables to identify such rule patterns (see [Mat09] for conditions of applicability for this rule).

Rule *b* establishes the commutative property of TSJ. Recall that, in Chapter 5, we proposed the evaluation of binary similarity joins by intersecting the two sorted inputs as we proceed with the algorithm. Due to its simplicity, this solution requires little modification on algorithms originally designed for self-similarity joins. Unfortunately, it can lead to suboptimal results when the join partners exhibit a marked difference in the number of input sets or set size distributions. For example, consider an extreme case where we join a set collection C_1 containing a single set of size *m* with a much larger set collection C_2. Further, suppose that $m > maxsize(x_{max})$, where x_{max} is the largest set of C_2. Clearly, the result of the set similarity join is empty. However, we would process the whole collection C_2 before we halt the algorithm and return the empty result. The development of a more efficient algorithm for binary joins is left open for future research. Nevertheless, the previous example shows that characteristics of each side of the join can affect performance; it also suggests that the problem can be tackled by adopting different processing strategies for each join input (like in hash-joins). In such cases, applying rewriting rules based on the commutative property is necessary.

We now consider the case for cost-based optimization. In Chapter 5, we evaluated set similarity joins employing the PCL_{slc} similarity function, which spans two profile representations. To this end, we adopted a multi-set representation and conjunctively combined the results derived from each profile. We can generalize this evaluation strategy for an arbitrary number of profiles whenever the corresponding similarity predicates are combined as conjunctive clauses, i.e, $\bigwedge sf_i \geq \tau_i$. This is important because efficient evaluation of conjunctive similarity predicates is a common demand to support EM applications. For example, decision rules are commonly expressed as a DNF formula [CCGK07], i.e., as a disjunction of conjuncts. In most of these cases, each similarity predicate relates to a different entity representation (different profiles or weighting schemes). Finally, while an arbitrary number of profiles leads to sets unbounded size, previous work has empirically observed that more than four conjuncts do not improve accuracy [CCGK07]. In such cases, multi-set representations of entities remain within the manageable size.

In Chapater 5, we observed identical results using structural and textual sets as primary sets. However, we may have several profiles defined over the content (or structural) part of XML documents and varying thresholds in similarity predicates. Therefore, the choice of the primary set in a multi-set representation is a crucial performance factor. Indeed, given that set similarity functions are computationally inexpensive, we can disregard the evaluation order of the remaining sets. Further, note that selecting the primary set corresponds to chosen the leftmost predicate in the conjunction—assuming that predicates are evaluated left-to-right order; accordingly, we call this predicate the primary predicate.

We now sketch a cost model for selecting the primary predicate based on based on the results reported in Chapter 5. We have seen that set size, set size distribution, feature frequency distribution skew, and threshold are important factors determining the cost of set similarity joins. In this regard, we can derive the following predicate cost model:

$$cost(s) = \frac{\varphi_1 \times ssize_n(p) + \varphi_2 \times sdistrib_n(p) + \varphi_3 \times fskew_n(p)}{\tau_p}\,, \tag{6.1}$$

where $ssize_n$, $sdeviation_n$, and $fskew_n$ are the normalized values corresponding to similarity predicate p for mean set size, standard deviation of set size distribution, and token frequency distribution skew; τ_p is the threshold of the predicate p; finally, we have $\varphi_1 + \varphi_2 + \varphi_3 = 1$. The primary predicate is therefore the predicate with the lowest cost according to the above formula. Devising an estimation model for the parameters is an interesting topic for future research.

Note that predicate selectivity is not explicitly considered in our cost model because has low influence on the overall runtime performance of our set similarity joins algorithms. Nevertheless, selectivity estimation is still needed when similarity joins are part of larger queries. Moreover, the observed Power-law behavior of token-frequency count distributions has been exploited to estimate the sizes of set similarity join outputs [LNS09], which suggests that there might exist a correlation between selectivity of a predicate and $fskew_n(.)$ In this respect, estimating the selectivity of TSJ and EDS queries also presents several research challenges. As far as we know, no research approach has tried so far to estimate the selectivity of XQuery expressions involving approximate path matching or selectivity of conjunctive similarity predicates (conditional independence assumption of path predicates may not hold). Finally, the work in [LNS09] is geared to unweighted sets; we are not aware of any work addressing selectivity estimation of set similarity joins for weighted sets.

6.7.3 Materialized Input Maintenance

We now discuss the maintenance of the materialized set similarity join input (materialized input, for short). It consists of a stored collection of token sets sorted in increasing order of the set sizes where each set is sorted in increasing order of token frequency. To update the materialized input when the underlying data changes, we can build a clustered B+-tree index on the sorted sets, with a composite key formed by the set size and DeweyID of the root tree node, both sorted in ascending order. Afterwards, tree updates can be propagated to the materialized input as follows: for insertions (deletions), the sorted set corresponding to the new (old) tree is generated and its size calculated; then, the set size and the root-node DeweyID are used to form a key record and the (old) new set is (deleted) inserted using the B+-tree; modifications comprise both insertion and deletion operations. Further, as for the auxiliary structures, updates can be buffered and propagated incrementally to the materialized input at fixed time intervals (updates on affected auxiliary structures precede updates on the materialized input). The target label used in the clustering process, the set of PCRs and DeweyIDs, and the PS are necessary for preprocessing the log of updates. Specifically, the following (interrelated) operations have to be performed:

- Filtering of operations that are not “covered” by the clustering process. As mentioned earlier, a node is covered by the clustering process if the path from the root to the node (including the node itself) contains at least one element node whose label is the same as the target label.
- Identification and grouping of node operations related to a single tree—the root node is determined by the topmost occurrence of the target label. Further, because some modifications may subsume or invalidate other operations, we can eliminate unnecessary operations. For example, a node modification in a tree can be followed by a deletion of the whole tree.
- Calculation of the root-node DeweyID for each tree identified by the previous operation.

With the actions above, we are able to update the materialized input regarding its constituent elements, i.e., sets and tokens within the sets, but not regarding the ordering and the weights of these elements. Updates on nodes or trees translate into modifications on the token-frequency table: new tokens can be inserted or existing tokens can be deleted or have their frequency increased or decreased. Further, for tree insertion or deletion, the size of the tree collection is increased or decreased. We discuss the effects of these updates as follows.

- *Token insertions and deletions*: Updates on the token-frequency table due to the insertion or deletion of tokens have no side-effects, because these tokens are only contained in the newly updated sets.

- *Tree insertions and deletions*: Alteration of the tree collection size changes the IDF weight of all tokens in the materialized input—recall that the size of the tree collection $|C|$ is used in the numerator of the IDF formula (see Equation 2.7 on Page 47). Hence, updating IDF weights to reflect a new $|C|$ value means re-building the materialized input from scratch. Fortunately, the logarithm in the IDF formula dampens the effect of $|C|$ value variations on token weights. Hence, unless the update batch has a highly unbalanced number of insertions and deletions, the IDF weights are only slightly changed and, in turn, degradation of accuracy is expected to be marginal.

- *Token-frequency modification*: Modification on the frequency of a token t may alter its relative ordering in all sets in which this token appears, i.e., in $\mathit{freq}(t, C)$ sets. Moreover, for weighted sets, the weight of the token changes (recall, $\mathit{freq}(t, C)$ is used in the denominator of the IDF formula) and, in turn, the weight of the set is changed accordingly. As a result, the relative set ordering may also be altered. Inconsistent token and set ordering can significantly jeopardize the algorithms presented in Chapter 5: valid results can be missed when matching tokens are not within prefixes and similar sets do not appear within the validity window of probing sets; conversely, misplaced tokens and sets can reduce filtering effectiveness. Finally, incorrect weights result in inaccurate similarity values.

Token insertions and deletions do not require our attention. In the following, we address updates changing size of the tree collection and token frequencies. Our discussion is based on the recent work of Hadjieleftheriou et al. [HKS09] in the context of index maintenance for set similarity selections.

As changes on the size of the tree collection have minor effects, we can tolerate some imprecision of the value of $|C|$. To this end, we define C_s as the value of $|C|$ used for weight calculation, when the materialized input is built. Furthermore, we specify a threshold for the maximum divergence between C_s and new values of $|C|$. Weight calculations due to updates are performed using C_s until the absolute value of $C_s - |C|$ exceeds the specified threshold; after that, the materialized input is rebuilt. For balanced tree insertions and deletions and reasonable thresholds, materialized input rebuilding can be considerably delayed.

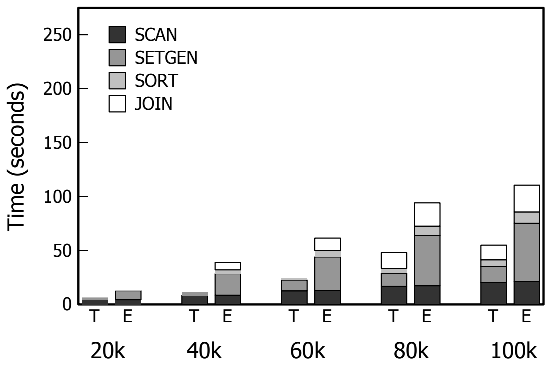
To propagate the new frequency of a token to the materialized input, we have to find and reorder all sets in which this token appears; for weight sets, we also need to calculate the new weight of t and update the set size; in turn, because the set size is part of the B+-tree index, we have to perform a deletion followed by an insertion. In this context, we need to construct inverted lists over all tokens to keep track of the containing sets for each token. Clearly, this solution requires substantial storage space and costly maintenance. Even worse, the effort required to perform updates related to highly frequent tokens on the materialized input approaches that of building it anew. Fortunately, high-frequency tokens have low IDF weight; hence, regarding the similarity value, they contribute to a lesser degree. Moreover, due to frequency ordering, such tokens are typically outside the prefixes. Thus, we can allow high-frequency values to vary within some range without degrading accuracy and efficiency too much. In a more general approach, the leeway for variation can increase proportionally to the token frequency. To this end, we can define a relaxation factor on the IDF value of a token—we can use this metric for unweighted sets as well. Similarly to the tree collection size, we define the value $idf_s(t)$ for each token t, which corresponds to the IDF weight at the builting time of the materialized input; this value can be stored in the token-frequency table (increasing the size of each entry by 8 bytes). Then, we update the materialized input due to frequency changes of a token t only when the divergence between its actual IDF weight and $idf_s(t)$ exceeds some predefined margin. Note that, owing to the logarithm of the IDF formula, much greater variation of high-frequency tokens is necessary to trigger updates on the materialized input. Alternatively, we can save both storage space and maintenance cost by not constructing inverted lists for high-frequency tokens. In this case, we rebuild the materialized input when the IDF of such tokens deviates outside the predefined threshold. Again, if the update behavior is balanced (regarding increases and decreases of token frequency), updates on the materialized input will be much less frequent.

In summary, our future work on this topic includes defining precise policies for incremental updates and—similarly to the work in [HKS09]—theoretically quantifying accuracy loss resulting from lazy update propagation.

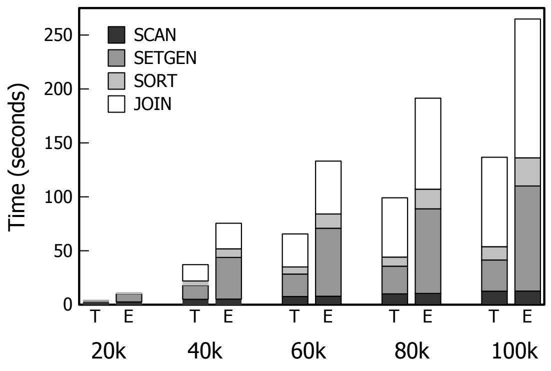
6.8 Experiments

We now evaluate the execution of the TSJ operator within XTC. Our main goal is to measure (i) the overall performance of the on-the-fly evaluation of the TSJ operator and (ii) the relative of performance of the execution steps described in Section 6.2.2, (iii) to compare the performance of TSJ using different similarity functions, and (iv) to examine the scalability of the TSJ components, in particular, those implementing steps 1–3 of the course of evaluation.

For the *Tree Access* step, we report runtime results relative to the TSJ input scan only (reported as SCAN in the experimental charts); the input selection evaluation is practically equivalent to evaluating regular XQuery expressions because the *NestRemOp* operator (recall TSJ QEP in Section 6.6) imposes negligible overhead. We refer the reader to [Mat09] for an extensive empirical assessment of XQuery processing in XTC. Further, we include the runtime results of Profile Generation and Set Generation (collectively reported as SETGEN), set collection sorting (SORT), and set similarity join



(a) SwissProt datasets with 20-100k trees



(b) PSD datasets with 20-100k trees

Figure 6.9: TSJ execution steps on an increasing number of trees using TLC and EPQ similarity functions

(JOIN). We evaluate TSJ using the similarity functions PCL_{tlc} and EPQ, which are abbreviated in the charts to T and E, respectively. We used the IDF weighting scheme and Jaccard as set similarity function.

We used similar datasets to those of Chapter 5: fuzzy copies of *SwissProt* and *PSD* datasets, where error injection parameters correspond to the values of *SwissProt-2M* and *PSD-2M* reported in the Table 4.2. To test scalability, we generated datasets varying from 20k to 100k, in steps of 20k. Hardware specifications are the same of Chapter 5. Finally, we fixed the threshold at 0.75.

The results are shown in Figure 6.9. On both datasets, SCAN, SETGEN, and SORT perfectly scale with the input size. Especially for SCAN, this fact indicates that we achieved seamless integration of similarity operators with regular XQuery processing operators. As expected from the results reported in Chapter 5, the runtime of JOIN grows quadratically as the dataset increases. SCAN is only slightly faster using TLC as compared to EPQ. As mentioned earlier, for generation of PCI-based profiles, we employ a simpler tree scan algorithm in which inner-node reconstruction is bypassed. Nevertheless, inner-node reconstruction results in little overhead because most of the computation involves DeweyID processing, in particular, LCA calculation, which is efficiently performed in XTC. Further, SCAN is about 80% faster on PSD (Figure 6.9b) as compared to SwissProt (Figure 6.9a) because characteristics of the PSD dataset lead to better compression rates of the storage representation. As a result, fewer disk blocks need to be read during the tree scan operation. On the other hand, SETGEN is about 2x slower on the PSD as compared to SwissProt for both similarity functions. The content part of PSD defined by the EDS queries is larger than those of SwissProt, which results in larger sets (see the set size distribution of PSD and SwissProt in Figure 5.9) and, in turn, higher workload for sorting and weighting operations. SETGEN is more than 3x faster on TLC as compared to EPQ. Because paths are provided for free by the path-oriented storage model, PCI-based token generation simply consists in accessing the PCR-PCI table and splitting strings into set of q-grams. On both datasets and for both similarity functions, SORT consumes only a small fraction of the overall processing time. The results of JOIN in isolation are identical to those reported in Chapter 5. In comparison to the other TSJ components, JOIN takes only up to 20% of the overall processing time on *SwissProt*, whereas it takes up to 60% on *PSD*. Because JOIN exhibits worse scalability than the others TSJ components, the fraction of TSJ processing owing to JOIN increases with the dataset size for both datasets. Of course, this fraction depends on data characteristics and the threshold parameter; for example, lowering the threshold would make JOIN processing the dominating factor for TSJ evaluation.

6.9 Related Work

To the best of our knowledge, our work is the first to investigate similarity joins in the context of XDBMSs. Our endeavor is inspired by the work of Chaudhuri et al. [CGK06], who proposed extending the set of physical operators to provide support for similarity joins inside relational database engines. Their similarity join operator, called SSJOIN, adopts the *unnested* set representation discussed in Chapter 5 and is composed of equi-joins, group-wise operators, and a few similarity-specific operators. This approach can leverage available cost-based optimizations of query engine to derive a cheap SSJoin implementation, e.g., based on index-based plans or hash-joins. Here, we implement set similarity joins using a single operator exploiting inverted lists [SK04] and explicitly "outsource" weighting schemes and tokenization methods to specific operators; hence, the composability feature of TSJ is geared towards flexibly providing a large space of similarity functions instead of various execution alternatives. Nevertheless, we can still benefit from algebraic optimizations and cost-based decisions as discussed earlier. Finally, our first version of TSJ followed the SSJoin implementation [RH07]; this version is outperformed by our current implementation based on inverted lists by orders of magnitude.

Early work on XML-based similarity joins was predominantly based on TED. Guha et al. [GJK+06] derived lower bounds by applying SED to the post-order representation of trees and upper bounds by imposing additional constraints on node relationships. Further, the authors used a pivot-based approach (pivots selected by sampling) to map trees into a vector space and indexed the resulting data representation using R-trees. Kailing et al. [KKSS04] applied histograms on several features of unordered trees, such as node degree and height, to save TED computations. Token-based approaches for XML similarity were first proposed by Yang et al. [YKT05] and Augsten et al. [ABG05].

Scoring and relevance ranking was recognized as a fundamental aspect of XML querying in [TW01]. Most approaches to flexible XML querying can be classified in two categories [VCÖ+07]. The first category is based on *query relaxation*, where structural transformations, such as axis generalization (i.e., changing "/" to "//") and

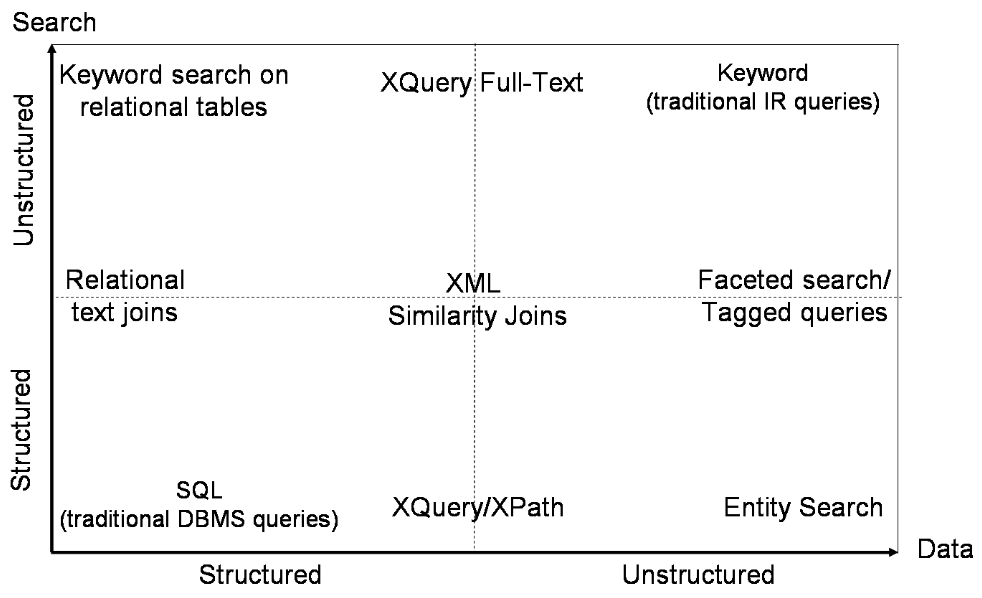


Figure 6.10: DB/IR query space (adapted from Weikum [Wei07])

keyword predicate moving, are applied to the original query to generate a new set of queries [AYLP04]. Weighting schemes are typically based on assessing the closeness between the matched pattern on the XML document and the original query. The second category returns closely related nodes, e.g., compact sub-trees containing all keywords, that are collectively relevant for the user query (e.g., [LYJ04]). Such approaches rely on heuristics or weighting schemes for selecting the most relevant combination of nodes. Besides scoring and relevance ranking, XML query processing has also been enhanced with other common IR engine features such as ontologies, query expansion, user feedback, and auto-completion [TBM+08, BW07]. Further, IR capabilities have been increasingly integrated into existing native XML databases [YQJ02] as well as hybrid commercial DBMSs [LM09] for which XQuery Full-Text support is provided. Other XML systems are designed from scratch for supporting ranked XML processing [TBM+08].

DB/IR integration is composed of many sub-areas, where each of them is already covered by a substantial amount of literature. In fact, this topic spans a rich solution space as data, queries, or both slide from structured to unstructured format. Here, we briefly describe the "coordinates" of XML similarity joins and elements in their neighborhood; we refer the interested reader to [CRW05, AYCR+05] and references therein for thorough overviews on DB/IR integration. Figure 6.10 shows elements of the query space supported by an integrated DB/IR architecture (adapted from Weikum [Wei07]). The x-axis represents the data format, while the y-axis classifies search queries; in both dimensions, the underlying format moves from structured to unstructured. As an invariant, stripping of structure from data or query translates into commuting from the exact matching paradigm to similarity matching. On the bottom-left corner, structured SQL queries appear, which are processed by traditional relational systems, whereas simple keyword queries processed by IR engines emerge at the opposite end of the diagonal. Removing the structure from queries on structured data, similarity joins over specific text fields and keyword queries over relational graphs occur. Adding structure to queries on unstructured data, queries are enhanced with content-oriented category metadata (faceted metadata), user-specified tags (e.g., from social networks), and search strategies that holistically identify entity concepts (a person, an event, etc) across several pages; therefore, they typically have to be backed by large-scale IE systems. Queries on semi-structured data can be classified placed to the middle part of Figure 6.10, i.e., structured XQuery/XPath queries on data-centric XML and full-text extension addressing document-centric XML. Finally, in the center of the solution space, the XML similarity joins are located, which encompass elements of all quadrants of the classification space.

6.10 Summary

This chapter puts the pieces of our XML similarity join framework together into an XDBMS. We were able not only to accommodate the components of our framework in the existing architecture but also to exploit XDBMS-specific features to optimize the realization of the similarity algorithms. We showed how to leverage indexing infrastructure, node identification mechanism, and physical algebra for XQuery processing to locate qualified XML fragments, to efficiently generate XML tree representations, and to compose pipelined query evaluation trees. In particular, the path-oriented storage model adopted in XTC was found to be a perfect match to the PCI-based similarity functions proposed in this thesis enabling the design of inexpensive algorithms for profile generation. We explicitly separated tokenization methods, weighting schemes, and set similarity calculation within the similarity join algorithm thereby allowing to easily plug-in different components or apply different parametrization thereof to serve EM applications with various notions of similarity. Further, we encapsulated these components into physical operators that can be flexibly assembled into query evaluation plans. This approach paves the way for employing algebraic optimizations and cost-based decisions to obtain better similarity join evaluation strategies. The performance and scalability of our solution was successfully validated experimentally. Finally, we addressed updatability of auxiliary structures and discussed further steps for providing tighter integration of our framework into XTC.

Chapter 7

Conclusions and Future Research

In this chapter, we conclude our work with a brief review of the problem addressed in this thesis and a summary of the contributions and results achieved, before we outline interesting research opportunities for future work.

7.1 Conclusions

This thesis presented the design and implementation of an effective, flexible, and high-performance XML-based similarity join framework. In particular, our framework aimed at supporting similarity joins on non-schematic and heterogeneous XML datasets. Performing similarity joins on XML data of this kind is challenging owing to several reasons. First, because the structure may vary along with textual information, the similarity function used in the join predicate has to address the hierarchical structure of XML trees and comparison results obtained from text and structure have to be pooled into a single notion of similarity. Further, structural heterogeneity also complicates the task of selecting the pieces of information considered for similarity assessment since related data may appear at different structural contexts among XML trees. Efficiency issues are also exacerbated on XML as similarity evaluation on tree-structured data is computationally intensive. Besides XML-motivated concerns, a similarity join framework has to flexibly support a rich variety of notions of similarity in order to serve primitive similarity operations to a wide range of applications.

Our framework addresses all the issues outlined above under reasonable data assumptions. The main contributions emanating from this work include novel *structure-conscious* similarity functions for XML trees, either considering XML structure in isolation or combined with textual information, mechanisms to support the selection of information represented by XML trees and organization of this information into a suitable format for similarity calculation, and efficient algorithms for large-scale identification of similar, set-represented objects. We validated our techniques in the context of a native XDBMS, which, in turn, spanned further contributions in the broader topic of marrying database and information retrieval technologies.

Two basic concepts provided the foundation for most of the techniques developed in this thesis. The first concept is that of *paths*, i.e., a sequence of hierarchically consecutive nodes in a tree. Important structural deviations found in heterogeneous XML datasets result in different paths within trees that encode and lead to the same information. We proposed a similarity function to find such paths and used a clustering method to group them; each cluster is represented by an integer called Path Cluster Identifier (PCI). By interpreting and representing path clusters as single units of information (tokens, in our nomenclature), we were able to derive compact structural surrogates for XML structures, which are bounded by the number of paths. More importantly, this representation delivered accurate results when incorporated into a similarity function, frequently outperforming more complex and expensive approaches. In this context, the so-called path synopsis, an index structure storing all paths appearing in an XML collection, was found very convenient for our approach: path synopses enabled us to adopt a ″one-for-all" approach, i.e., instead of computing anew the similarity between repeated structures during join execution, we compute the similarity between all paths in a path synopsis only once in a pre-processing stage. Besides providing structural representation, PCIs also play a pivotal role in the interplay between text and structure for similarity assessment. First, PCIs can be prepended to textual tokens to restrict the comparison of text nodes to those appearing in similar structural contexts. Second, the delimitation between structural and textual representation of XML trees is based on decomposing the set of PCIs into two mutually exclusive subsets. We let the user define this decomposition by issuing the so-called EDS queries, i.e., simple path specifications that are approximately matched against the set of clusters; the PCIs of the most similar path clusters define the textual representation, whereas the remaining PCIS define the structural representation. Finally, to efficiently evaluate EDS queries, we summarize path clusters into concise structures represented as short memory-resident inverted lists and ensure truly interactive response times by employing well-known IR optimizations.

The second foundational concept appearing in many places of this thesis is that of *sets*. While paths were a central piece of information of XML trees in our approach, sets provided a simple yet powerful representation for paths as well as for strings and whole XML trees. By operating on sets instead of the original representations, we reduced the similarity calculation to the set-overlap problem, where different ways of measuring the overlap of sets raise several notions of similarity. This approach brought several advantages. We could measure textual and structural similarity between XML trees, jointly or in isolation, by operating on sets representing text, structure, or both, in a unified framework. We also obtained a very rich similarity space by varying the methods for mapping XML trees to sets, associating weights to set elements (weighting schemes), measuring the set overlap, or any combination thereof. Besides being inexpensive to calculate, the set-overlap abstraction gives raise to several optimizations for pruning the comparison space of similarity joins. Building upon these optimizations, but following an opposite approach w.r.t. previous work, we presented a new algorithm that consistently outperformed state-of-the-art set similarity join algorithms.

Finally, the integration of our framework into an XDBMS architecture brought to the context of our thesis several aspects of core database technology. In this context, we exploited XDBMS-specific features to optimize the realization of the similarity joins algorithms proposed in this thesis. Specifically, we leveraged storage model, indexing infrastructure, node identification mechanism, and physical algebra for XQuery processing to locate qualified XML fragments, to efficiently generate XML tree representations, and to compose pipelined query evaluation trees. In particular, we found the path-oriented storage model a perfect match to our PCI-based based approach, in which we were able to derive path-based XML representation practically for free.

Although we have situated our work in the context of identifying the so-called fuzzy duplicates, most techniques proposed here easily reduce to the general problem of identifying similar tree-structured information in large datasets. Hence, the contributions of this thesis can be straightforwardly conveyed to other application domains. Owing to the increasingly ubiquitous concept of similarity, there is a very wide spectrum of such potential domains, including data mining, computational biology, Web indexing, social networks, information extraction, pattern recognition, collaborative filtering, just to name a few. Moreover, while the techniques presented in Chapter 3 and 4 are geared for specific aspects of XML data, the set similarity join algorithms introduced in Chapter 5 are applicable to any object and data types that lend themselves to set representation. Likewise, in our discussion on the integration of our framework into an XDBMS presented in Chapter 6, we touched several aspects that are also relevant to DB/IR integration in general.

7.2 Future Research

Of course, we could not cover the topic of this thesis completely. Hence, there are many directions for future work. Perhaps, the most intriguing of them is providing tighter integration of our framework into an XDBMS architecture. In Section 6.7, we have already discussed in great details our future steps in this direction, namely, extensions to existing query languages, integration into a query optimizer, and materialized input maintenance.

Another interesting line of research relates to evaluating effectiveness of structural similarity functions. In this thesis, we evaluated effectiveness experimentally. Similar to other work [ZM98, CRF03] in the context of IR and similarity matching on strings, we identified no ”silver bullet“, i.e., no structural similarity function was overall the best across all datasets. This result was highly expected. All similarity functions are able to capture certain data deviations while ignoring others; moreover, deviations can be weighed differently. Hence, the effectiveness of a similarity function depends on the agreement between its ”abilities“ and the characteristics of duplicates and non-duplicates — or, more generally, characteristics of similar and dissimilar objects — in a dataset w.r.t. a given application or task. In theory, given a precise enumeration of the behavior of a similarity function for a representative set of structural patterns and a statistical characterization of the deviations present in duplicates and non-duplicates, it would be possible to predict the performance of this similarity function and even deem this similarity function as better or more appropriate than others in a given context. In practice, learning models are employed for tuning, finding a set of transformations, and defining a combination and weighting strategy of similarity functions, [BM03a, SB02, TKM02, ACK09]. However, very little attention is given to the identification and quantification of the characteristics of datasets and similarity functions that lead to the effectiveness results. For text, this task can be overwhelming because the ambit of deviations is very large. On the other hand, in comparison to text, structure is expected to deviate much more moderately and, therefore, characterization of structural deviations on some datasets can be feasible. Some similarity functions have known properties regarding their structural sensitivity, such as marked sensitivity to node fanout (e.g, *pq*-grams [ABG10]), ability in detecting node moves to other parents (e.g., windowed pq-grams [ABDG08]) and inversion of nodes in a path (e.g, our own PCI-based similarity function). However, relatively little is known, for example, about other structural patterns that can negatively affect the performance of these techniques. Thus, we believe that more progress can be made on predicting the performance of similarity functions by systematically characterizing their behavior on common classes of structural deviations.

We also see much room for optimization of the techniques presented in this thesis. Although yielding good accuracy results, our path clustering strategy sometimes creates spurious clusters in the sense that several unrelated paths are grouped. While structural similarity assessment is to some extent resilient to spurious clusters, comparison of values from unrelated concepts can severely jeopardize the results for textual similarity. Therefore, it would be desirable to devise procedures to evaluate the validity (or purity) of path clusters in a quantitative manner and assist the user in correcting misplaced paths. Moreover, considering that the clustering process is performed off-line and, therefore, efficiency requirements are relatively more flexible, we can (substantially) increase accuracy by exploiting text node values to identify path duplicates instead of solely relying on element node labels. To this end, we can build upon previous work on mining the structure of relational databases based on field values [DJMS02].

Further, our framework requires several parameters, such as cutting point threshold for the dendogram and decay rate of the path similarity function. Identifying the best configuration setting can be time-consuming. For this reason, we plan to investigate techniques based on sampling or learning models to obtain optimal parameters semi-automatically. We also would like to support more expressive EDS queries (for example, queries containing branches) and enhance user guidance on formulation of EDS queries with information-theoretic techniques [Seb02].

Finally, it would be desirable to improve our set similarity join algorithms for binary joins, to enhance the presentation of similarity join results by incorporating result post-processing into our framework, and to integrate strategies for parallel processing into our framework.

Bibliography

[ABC99] M. Arenas, L. Bertossi, and J. Chomicki, "Consistent query answers in inconsistent databases," in *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1999)*, 1999.

[ABDG08] N. Augsten, M. H. Böhlen, C. E. Dyreson, and J. Gamper, "Approximate joins for data-centric xml," in *Proc. of the 24th Intl. Conf. on Data Engineering (ICDE 2008)*, 2008, pp. 814–823.

[ABG05] N. Augsten, M. H. Böhlen, and J. Gamper, "Approximate matching of hierarchical data using pq-grams," in *Proc. of the 31st Intl. Conf. on Very Large Data Bases (VLDB 2005)*. ACM, 2005, pp. 301–312.

[ABG06] N. Augsten, M. H. Böhlen, and J. Gamper, "An incrementally maintainable index for approximate lookups in hierarchical data," in *Proc. of the 32nd Intl. Conf. on Very Large Data Bases (VLDB 2006)*. ACM, 2006, pp. 247–258.

[ABG10] N. Augsten, M. H. Böhlen, and J. Gamper, "The *q*-gram distance between ordered labeled trees," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 1, 2010.

[ACG02] R. Ananthakrishna, S. Chaudhuri, and V. Ganti, "Eliminating fuzzy duplicates in data warehouses," in *Proc. of 28th Intl. Conf. on Very Large Data Bases (VLDB 2002)*, 2002, pp. 586–597.

[ACK08] A. Arasu, S. Chaudhuri, and R. Kaushik, "Transformation-based framework for record matching," in *Proc. of the 24th Intl. Conf. on Data Engineering (ICDE 2008)*, 2008, pp. 40–49.

[ACK09] A. Arasu, S. Chaudhuri, and R. Kaushik, "Learning string transformations from examples," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 514–525, 2009.

[AFM06] P. Andritsos, A. Fuxman, and R. J. Miller, "Clean answers over dirty databases: A probabilistic approach," in *Proc. of the 22nd Intl. Conf. on Data Engineering (ICDE 2006)*, 2006, p. 30.

[AGK06] A. Arasu, V. Ganti, and R. Kaushik, “Efficient exact set-similarity joins,” in *Proc. of the 32nd Intl. Conf. on Very Large Data Bases (VLDB 2006)*, 2006, pp. 918–929.

[AKJP^{+}02] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava, “Structural joins: A primitive for efficient xml query pattern matching,” in *Proc. of the 18th Intl. Conf. on Data Engineering*, 2002, pp. 141–.

[ATW^{+}07] C. C. Aggarwal, N. Ta, J. Wang, J. Feng, and M. J. Zaki, “Xproj: a framework for projected structural clustering of xml documents,” in *Proc. of the 13th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2007, pp. 46–55.

[AYBB^{+}10] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, M. Dyck, M. Holstege, J. Melton, M. Rys, and J. Shanmugasundaram, “Xquery and xpath full text 1.0,” 2010, http://www.w3.org/TR/xpath-full-text-10/.

[AYBS04] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram, “Texquery: a full-text search extension to xquery,” in *Proc. of the 13th Intl. Conf. on World Wide Web (WWW 2004)*, 2004, pp. 583–594.

[AYCR^{+}05] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum, “Report on the db/ir panel at sigmod 2005,” *SIGMOD Record*, vol. 34, no. 4, pp. 71–74, 2005.

[AYL06] S. Amer-Yahia and M. Lalmas, “Xml search: languages, inex and scoring,” *SIGMOD Record*, vol. 35, no. 4, pp. 16–23, 2006.

[AYLP04] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit, “Flexpath: Flexible structure and full-text querying for xml,” in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2004, pp. 83–94.

[AYMH^{+}08] S. Amer-Yahia, V. Markl, A. Y. Halevy, A. Doan, G. Alonso, D. Kossmann, and G. Weikum, “Databases and web 2.0 panel at vldb 2007,” *SIGMOD Record*, vol. 37, no. 1, pp. 49–52, 2008.

[Baa01] R. H. Baayen, *Word Frequency Distributions*. Kluwer Academic Publishers, 2001.

[BBC04] N. Bansal, A. Blum, and S. Chawla, “Correlation clustering,” *Machine Learning*, vol. 56, no. 1-3, pp. 89–113, 2004.

[BBC^{+}07] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernndez, M. Kay, J. Robie, and J. Simon, “Xml path language (xpath) 2.0,” 2007, http://www.w3.org/TR/xpath20.

[BBS05] M. Bilenko, S. Basu, and M. Sahami, "Adaptive product normalization: Using online learning for record linkage in comparison shopping," in *Proc. of the 5th IEEE Intl. Conf. on Data Mining (ICDM 2005)*, 2005, pp. 58–65.

[BCC03] R. Baxter, P. Christen, and T. Churches, "A comparison of fast blocking methods for record linkage," in *Proc. of the Workshop on Data Cleaning, Record Linkage and Object Consolidation at the 9th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2003.

[BCF$^+$07] S. Boag, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, and J. Simon, "Xquery 1.0: An xml query language," 2007, http://www.w3.org/TR/xquery.

[BCJ$^+$05] K. S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, R. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. C. Truong, B. V. der Linden, B. Vickery, and C. Zhang, "System rx: One part relational, one part xml," in *Proc. SIGMOD Conf.*, 2005, pp. 347–358.

[BDF$^+$97] D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik, "The new jersey data reduction report," *IEEE Data Engineering Bulletin*, vol. 20, no. 4, pp. 3–45, 1997.

[BEF08] B. Bryan, F. Eberhardt, and C. Faloutsos, "Compact similarity joins," in *Proc. of the 24th Intl. Conf. on Data Engineering (ICDE 2008)*, 2008, pp. 346–355.

[Ber01] M. K. Bergman, "The deep web: Surfacing hidden value," *Journal of Electronic Publishing*, vol. 7, no. 1, 2001.

[BFFR05] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi, "A cost-based model and effective heuristic for repairing constraints by value modification," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2005)*, 2005, pp. 143–154.

[BFG$^+$07] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for data cleaning," in *Proc. of the 23rd Intl. Conf. on Data Engineering (ICDE 2007)*, 2007, pp. 746–755.

[BG06] I. Bhattacharya and L. Getoor, "A latent dirichlet model for unsupervised entity resolution," in *Proc. of the 6th SIAM Intl. Conf. on Data Mining (SDM 2006)*, 2006.

[BGK03] P. Buneman, M. Grohe, and C. Koch, "Path queries on compressed xml," in *Proc. of 29th Intl. Conf. on Very Large Data Bases (VLDB 2003)*, 2003, pp. 141–152.

[BGL$^+$98] C. H. Bennett, P. Gács, M. Li, P. M. B. Vitányi, and W. H. Zurek, “Information distance,” *IEEE Transactions on Information Theory*, vol. 44, no. 4, pp. 1407–1423, 1998.

[BGMM$^+$09] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom, “Swoosh: a generic approach to entity resolution,” *The VLDB Journal*, vol. 18, no. 1, pp. 255–276, 2009.

[BGRS99] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, “When is ”nearest neighbor” meaningful?” in *International Conference on Database Theory (ICDT’99)*, 1999, pp. 217–235.

[Bis06] C. M. Bishop, “Pattern recognition and machine learning.” Springer, 2006.

[BKM06] M. Bilenko, B. Kamath, and R. J. Mooney, “Adaptive blocking: Learning to scale up record linkage,” in *Proc. of the 6th IEEE Intl. Conf. on Data Mining (ICDM 2006)*, 2006, pp. 87–96.

[BKS02] N. Bruno, N. Koudas, and D. Srivastava, “Holistic twig joins: optimal xml pattern matching,” in *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2002)*, 2002, pp. 310–321.

[BM03a] M. Bilenko and R. J. Mooney, “Adaptive duplicate detection using learnable string similarity measures,” in *Proc. of the 9th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (KDD 2003)*, 2003, pp. 39–48.

[BM03b] M. Bilenko and R. J. Mooney, “On evaluation and training-set construction for duplicate detection,” in *Proc. KDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003, pp. 7–12.

[BMS07] R. J. Bayardo, Y. Ma, and R. Srikant, “Scaling up all pairs similarity search,” in *Proc. of the 16th Intl. Conf. on World Wide Web (WWW 2007)*, 2007, pp. 131–140.

[BN05] A. Bilke and F. Naumann, “Schema matching using duplicates,” in *Proc. of the 21st Intl. Conf. on Data Engineering (ICDE 2005)*, 2005, pp. 69–80.

[BN08] J. Bleiholder and F. Naumann, “Data fusion,” *ACM Computing Surveys (CSUR)*, vol. 41, no. 1, 2008.

[BNJ03] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[BP09] N. L. Bhamidipati and S. K. Pal, "Comparing scores intended for ranking," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 21, no. 1, pp. 21–34, 2009.

[Bro97] A. Z. Broder, "On the resemblance and containment of documents," in *Proc. of the Intl. Conf. on Compression and Complexity of Sequences(SEQUENCES'97)*, 1997, pp. 21–29.

[BSH$^+$08] O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald, and J. Widom, "Databases with uncertainty and lineage," *The VLDB Journal*, vol. 17, no. 2, pp. 243–264, 2008.

[BSIBD09] G. Beskales, M. A. Soliman, I. F. Ilyas, and S. Ben-David, "Modeling and querying possible repairs in duplicate detection," *Proc. of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 598–609, 2009.

[But04] D. Buttler, "A short survey of document structure similarity algorithms," in *Proc. of the Intl. Conf. on Internet Computing (IC'04)*. CSREA Press, 2004, pp. 3–9.

[BW07] H. Bast and I. Weber, "The completesearch engine: Interactive, efficient, and towards ir& db integration," in *Proc. of the 3rd Biennial Conf. on Innovative Data Systems Research (CIDR 2007)*, 2007, pp. 88–95.

[CAM02] G. Cobena, S. Abiteboul, and A. Marian, "Detecting changes in xml documents," in *Proc. of the 19th Intl. Conf. on Data Engineering (ICDE 2003)*. IEEE Computer Society, 2002, pp. 41–52.

[CCGK07] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik, "Example-driven design of efficient record matching queries," in *Proc. of the 33rd Intl. Conf. on Very Large Data Bases (VLDB 2007)*, 2007, pp. 327–338.

[CCGX08] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin, "An efficient filter for approximate membership checking," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2008)*, 2008, pp. 805–818.

[CCX08] R. Cheng, J. Chen, and X. Xie, "Cleaning uncertain data with quality guarantees," *Proc. of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 722–735, 2008.

[CD09] S. Chaudhuri and G. Das, "Keyword querying and ranking in databases," *PVLDB*, vol. 2, no. 2, pp. 1658–1659, 2009.

[CDF$^+$01] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Mootwani, J. D. Ullman, and C. Yang, "Finding interesting associations without support pruning," *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, vol. 13, no. 1, pp. 64–78, 2001.

[CDHW06] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum, "Probabilistic information retrieval approach for ranking of database query results," *TODS*, vol. 31, no. 3, pp. 1134–1168, 2006.

[CFG+07] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: Consistency and accuracy," in *Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, 2007, pp. 315–326.

[CGK06] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *Proc. of the 22nd Intl. Conf. on Data Engineering (ICDE 2006)*, 2006, p. 5.

[CGM97] S. S. Chawathe and H. Garcia-Molina, "Meaningful change detection in structured data," in *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 1997)*. ACM, 1997, pp. 26–37.

[CGM05] S. Chaudhuri, V. Ganti, and R. Motwani, "Robust identification of fuzzy duplicates," in *Proc. of the 21st Intl. Conf. on Data Engineering (ICDE 2005)*, 2005, pp. 865–876.

[CGS03] S. Chaudhuri, P. Ganesan, and S. Sarawagi, "Factorizing complex predicates in queries to exploit indexes," in *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, 2003, pp. 361–372.

[Cha02] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proc. on 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*, 2002, pp. 380–388.

[CHK+07] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava, "Benchmarking declarative approximate selection predicates," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2007)*, 2007, pp. 353–364.

[CHW+08] M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. Wu, and Y. Zhang, "Webtables: Exploring the power of tables on the web," *PVLDB*, vol. 1, no. 1, pp. 538–549, 2008.

[CK09] S. Chaudhuri and R. Kaushik, "Extending autocompletion to tolerate errors," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2009)*, 2009, pp. 707–718.

[CKVW06] D. Cheng, R. Kannan, S. Vempala, and G. Wang, "A divide-and-merge methodology for clustering," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 4, pp. 1499–1525, 2006.

[CMM+03] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer, "Searching xml documents via xml fragments," in *SIGIR 2003: Proc.*

of the 26th Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR 2003), 2003, pp. 151–158.

[CNBYM01] E. Chávez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Computing Surveys (CSUR)*, vol. 33, no. 3, pp. 273–321, 2001.

[CNS04] S. Chaudhuri, V. R. Narasayya, and S. Sarawagi, "Extracting predicates from mining models for efficient query evaluation," *ACM Transactions on Database Systems (TODS)*, vol. 29, no. 3, pp. 508–544, 2004.

[Coh98] W. W. Cohen, "Integration of heterogeneous databases without common domains using queries based on textual similarity," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1998, pp. 201–212.

[Coh04] J. Cohen, "Bioinformatics - an introduction for computer scientists," *ACM Computing Surveys (CSUR)*, vol. 36, no. 2, pp. 122–158, 2004.

[CRF03] W. W. Cohen, P. D. Ravikumar, and S. E. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *Proc. of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03)*, 2003, pp. 73–78.

[CRGMW96] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proc. of the ACM 1996 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 1996)*. ACM, 1996, pp. 493–504.

[Cro00] W. B. Croft, "Combining approaches to information retrieval," *Advances in information retrieval*, vol. 7, pp. 1–36, 2000.

[CRW05] S. Chaudhuri, R. Ramakrishnan, and G. Weikum, "Integrating db and ir technologies: What is the sound of one hand clapping?" in *Proc. CIDR Conf.*, 2005, pp. 1–12.

[CSGK07] S. Chaudhuri, A. D. Sarma, V. Ganti, and R. Kaushik, "Leveraging aggregate constraints for deduplication," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2007)*, 2007, pp. 437–448.

[CSN09] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Review*, vol. 51, no. 4, pp. 661–703, 2009.

[CT06] T. M. Cover and J. A. Thomas, *Elements of information theory*. Wiley, 2006.

[CV05] R. Cilibrasi and P. M. B. Vitányi, "Clustering by compression," *IEEE Transactions on Information Theory*, vol. 51, no. 4, pp. 1523–1545, 2005.

[CVDN09] X. Chai, B.-Q. Vuong, A. Doan, and J. F. Naughton, "Efficiently incorporating user feedback into information extraction and integration programs," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2009)*, 2009, pp. 87–100.

[CYC07] T. Cheng, X. Yan, and K. C.-C. Chang, "Entityrank: Searching entities directly and holistically," in *Proc. VLDB Conf.*, 2007, pp. 387–398.

[dAMF10] J. de Aguiar Moraes Filho, "Summarizing xml documents: Contributions, empirical studies, and challenges," Ph.D. dissertation, Technische Universität Kaiserslautern, 2010.

[dbl09] "The dblp computer science bibliography," 2009, http://dblp.uni-trier.de/xml.

[DCWS06] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. K. Sellis, "A methodology for clustering xml documents by structure," *Information Systems*, vol. 31, no. 3, pp. 187–228, 2006.

[DDL+90] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science and Technology (JASIST)*, vol. 41, no. 6, pp. 391–407, 1990.

[DFS+09] E. C. Dragut, F. Fang, A. P. Sistla, C. T. Yu, and W. Meng, "Stop word and related problems in web interface integration," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 349–360, 2009.

[DG09] L. Denoyer and P. Gallinari, "Overview of the inex 2008 xml mining track," in *In Proc. of INEX 2008*, 2009.

[DJMS02] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk, "Mining database structure; or, how to build a data quality browser," in *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, 2002, pp. 240–251.

[DKP+09] N. N. Dalvi, R. Kumar, B. Pang, R. Ramakrishnan, A. Tomkins, P. Bohannon, S. Keerthi, and S. Merugu, "A web of concepts," in *Proc. of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2009)*, 2009, pp. 1–12.

[DMRW07] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann, "An optimal decomposition algorithm for tree edit distance," in *Proc. of the 34th Intl. Colloquium on Automata, Languages and Programming (ICALP 2007)*, 2007, pp. 146–157.

[DNH$^+$09] C. F. Dorneles, M. F. Nunes, C. A. Heuser, V. P. Moreira, A. S. da Silva, and E. S. de Moura, "A strategy for allowing meaningful and comparable scores in approximate matching," *Information Systems*, vol. 34, no. 8, pp. 673–689, 2009.

[Dop08] P. Dopichaj, "Content-oriented retrieval on document-centric xml," Ph.D. dissertation, Technische Universität Kaiserslautern, 2008.

[DRS09] N. N. Dalvi, C. Ré, and D. Suciu, "Probabilistic databases: Diamonds in the dirt," *Communications of the ACM (CACM)*, vol. 52, no. 7, pp. 86–94, 2009.

[DT03] S. Dulucq and H. Touzet, "Analysis of tree edit distance algorithms," in *Proc. of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, ser. Lecture Notes in Computer Science, vol. 2676. Springer, 2003, pp. 83–95.

[EEV02] M. G. Elfeky, A. K. Elmagarmid, and V. S. Verykios, "Tailor: A record linkage tool box," in *Proc. of the 18th Intl. Conf. on Data Engineering (ICDE 2002)*, 2002, pp. 17–28.

[EIV07] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *TKDE*, vol. 19, no. 1, pp. 1–16, 2007.

[Fel98] C. Fellbaum, *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[FGKL02] N. Fuhr, N. Gövert, G. Kazai, and M. Lalmas, Eds., *INEX Workshop 2002*, 2002.

[FKS03] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," in *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, 2003, pp. 28–36.

[FL95] C. Faloutsos and K.-I. Lin, "Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets," in *Proc. of the 1995 ACM SIGMOD Intl. Conf. on Management of Data*, 1995, pp. 163–174.

[FMM$^+$05] S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese, "Fast detection of xml structural similarity," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 17, no. 2, pp. 160–175, 2005.

[FS69] I. P. Fellegi and A. B. Sunter, "A theory for record linkage," *Journal of the American Statistical Association (JASA)*, vol. 64, no. 328, pp. 1183–1210, 1969.

[FSN$^+$95] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker, "Query by image and video content: The qbic system," *Computer*, vol. 28, no. 9, pp. 23–32, 1995.

[GdSM07] R. Gonçalves and R. dos Santos Mello, "Improving xml instances comparison with preprocessing algorithms," in *Proc. of the 18th Intl. Conf. on Database and Expert Systems Applications (DEXA 2007)*, 2007, pp. 13–22.

[GFS$^+$01] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita, "Declarative data cleaning: Language, model, and algorithms," in *Proc. of 27th Intl. Conf. on Very Large Data Bases (VLDB 2001)*, 2001, pp. 371–380.

[GIJ$^+$01] 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 *Proc. of 27th Intl. Conf. on Very Large Data Bases (VLDB 2001)*, 2001, pp. 491–500.

[GIKS03] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava, "Text joins in an rdbms for web data integration," in *Proc. of the 12th Intl. World Wide Web Conf. (WWW 2003)*, 2003, pp. 90–101.

[GIM99] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. of 25th Intl. Conf. on Very Large Data Bases (VLDB'99)*, 1999, pp. 518–529.

[GJK$^+$06] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu, "Integrating xml data sources using approximate joins," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 161–207, 2006.

[GKMS04] S. Guha, N. Koudas, A. Marathe, and D. Srivastava, "Merging the results of approximate match operations," in *Proc. of the 13th Intl. Conf. on Very Large Data Bases*, 2004, pp. 636–647.

[Gör10] J. Göres, "A model management framework for information integration," Ph.D. dissertation, Technische Universität Kaiserslautern, 2010.

[Gra93] G. Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys (CSUR)*, vol. 25, no. 2, pp. 73–170, 1993.

[GSE$^+$94] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," 1994, pp. 243–252.

[GW97] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," 1997, pp. 436–445.

[Hau05] M. M. Haustein, “Feingranulare transaktionsisolation in nativen xml-datenbanksystemen,” Ph.D. dissertation, Technische Universität Kaiserslautern, 2005.

[HCKS08] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, “Fast indexes and algorithms for set similarity selection queries,” in *Proc. of the 24th Intl. Conf. on Data Engineering (ICDE 2008)*, 2008, pp. 267–276.

[HCML09] O. Hassanzadeh, F. Chiang, R. J. Miller, and H. C. Lee, “Framework for evaluating clustering algorithms in duplicate detection,” *Proc. of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 1282–1293, 2009.

[Hel07] S. Helmer, “Measuring the structural similarity of semistructured documents using entropy,” in *Proc. of the 33rd Intl. Conf. on Very Large Data Bases (VLDB 2007)*. ACM, 2007, pp. 1022–1032.

[HFM06] A. Y. Halevy, M. J. Franklin, and D. Maier, “Principles of dataspace systems,” in *Proc. 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2006)*, 2006, pp. 1–9.

[HHH$^+$] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth, “Clio grows up: from research prototype to industrial tool,” in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2005).*

[HHMW07] T. Härder, M. P. Haustein, C. Mathis, and M. Wagner, “Node labeling schemes for dynamic xml documents reconsidered,” *Data Knowledge Engineering*, vol. 60, no. 1, pp. 126–149, 2007.

[HKS09] M. Hadjieleftheriou, N. Koudas, and D. Srivastava, “Incremental maintenance of length normalized indexes for approximate string matching,” in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2009)*, 2009, pp. 429–440.

[HMS07] T. Härder, C. Mathis, and K. Schmidt, “Comparison of complete and elementless native storage of xml documents,” in *Proc. of the 11th Intl. Database Engineering and Applications Symposium (IDEAS 2007)*, 2007, pp. 102–113.

[Hof01] T. Hofmann, “Unsupervised learning by probabilistic latent semantic analysis,” *Machine Learning*, vol. 42, no. 1/2, pp. 177–196, 2001.

[HR83] T. Härder and A. Reuter, “Concepts for implementing a centralized database management system,” in *Proc. of the Intl. Computing Symposium on Application Systems Development*, 1983, pp. 28–60.

[HS93] J. M. Hellerstein and M. Stonebraker, “Predicate migration: Optimizing queries with expensive predicates,” in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 1993)*, 1993, pp. 267–276.

[HS98] M. A. Hernández and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," *Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 9–37, 1998.

[HYKS08] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava, "Hashed samples: selectivity estimators for set similarity selection queries," *Proc. of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 201–212, 2008.

[III09] I. Information Impact International, "Publicly exposed iq problems," http://www.infoimpact.com/publiclyexposediqproblems.cfm, 2009.

[JAKN03] S. Joshi, N. Agrawal, R. Krishnapuram, and S. Negi, "A bag of paths model for measuring structural similarity in web documents," in *Proc. of the 9th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2003, pp. 577–582.

[JCE$^+$07] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu, "Making database systems usable," in *Proc. SIGMOD Conf.*, 2007, pp. 13–24.

[JD88] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*. Prentice Hall College Div, 1988.

[JFH08] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy, "Pay-as-you-go user feedback for dataspace systems," 2008, pp. 847–860.

[JLM03] L. Jin, C. Li, and S. Mehrotra, "Efficient record linkage in large data sets," in *Proc. of the 8th Intl. Conf. on Database Systems for Advanced Applications (DASFAA'03)*, 2003, p. 137.

[Jon72] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[JS07] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Trans. on Database Systems (TODS)*, vol. 32, no. 1, p. 7, 2007.

[JS08] E. H. Jacox and H. Samet, "Metric space similarity joins," *ACM Trans. on Database Systems (TODS)*, vol. 33, no. 2, 2008.

[KKSS04] K. Kailing, H.-P. Kriegel, S. Schönauer, and T. Seidl, "Efficient similarity search for hierarchical data in large databases," in *Proc. of the Intl. Conf. on Extending Database Technology (EDBT 2004)*, 2004, pp. 676–693.

[Kle98] P. N. Klein, "Computing the edit-distance between unrooted ordered trees," in *Proc. of the 6th Annual European Symposium on Algorithms (ESA 1998)*, 1998, pp. 91–102.

[KMdRS06] J. Kamps, M. Marx, M. de Rijke, and B. Sigurbjörnsson, "Articulating information needs in xml query languages," *ACM Transactions on Information Systems (TOIS)*, vol. 24, no. 4, pp. 407–436, 2006.

[Kol05] P. G. Kolaitis, "Schema mappings, data exchange, and metadata management," in *Proc. of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2005)*, 2005, pp. 61–75.

[KR87] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[KSS06] N. Koudas, S. Sarawagi, and D. Srivastava, "Record linkage: Similarity measures and algorithms," in *Proc. SIGMOD Conf.*, 2006, pp. 802–803.

[Kuk92] K. Kukich, "Techniques for automatically correcting words in text," *ACM Computing Surveys (CSUR)*, vol. 24, no. 4, pp. 377–439, 1992.

[LCBC08] Z. H. Liu, S. Chandrasekar, T. Baby, and H. J. Chang, "Towards a physical xml independent xquery/sql/xml engine," *PVLDB*, vol. 1, no. 2, pp. 1356–1367, 2008.

[LCMY04] W. Lian, D. W.-L. Cheung, N. Mamoulis, and S.-M. Yiu, "An efficient and scalable algorithm for clustering xml documents by structure," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 16, no. 1, pp. 82–96, 2004.

[LCW07] L. Leitão, P. Calado, and M. Weis, "Structure-based inference of xml similarity for fuzzy duplicate detection," in *Proc. of the 6th ACM Conf. on Information and Knowledge Management (CIKM 2007)*, 2007, pp. 293–302.

[Lee97] J. H. Lee, "Analyses of multiple evidence combination," in *Proc. of the 20th Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR 97)*, 1997, pp. 267–276.

[Len05] R. Lenz, "Information management in distributed healthcare networks," in *Data Management in a Connected World*, ser. LNCS, vol. 3551, 2005, pp. 315–334.

[Lev65] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Doklady Akademii Nauk SSSR*, vol. 163, no. 4, pp. 845–848, 1965.

[LLL08] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *Proc. of the 24th Intl. Conf. on Data Engineering (ICDE 2008)*, 2008, pp. 257–266.

[LM09] Z. H. Liu and R. Murthy, “A decade of xml data management: An industrial experience report from oracle,” in *Proc. ICDE Conf.*, 2009, pp. 1351–1362.

[LMP01] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, “Coonditional random fields: Probabilistic models for segmenting and labeling sequence data,” in *International Conference on Machine Learning (ICML)*, 2001, pp. 282–289.

[LNS09] H. Lee, R. T. Ng, and K. Shim, “Power-law based estimation of set similarity join size,” *Proc. of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 658–669, 2009.

[LSS08] M. D. Lieberman, J. Sankaranarayanan, and H. Samet, “A fast similarity join algorithm using graphics processing units,” in *Proc. of the 24th Intl. Conf. on Data Engineering (ICDE 2008)*, 2008, pp. 1111–1120.

[LYJ04] Y. Li, C. Yu, and H. V. Jagadish, “Schema-free xquery,” 2004, pp. 72–83.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, “Lore: A database management system for semistructured data,” *SIGMOD Record*, vol. 26, no. 3, pp. 54–66, 1997.

[Mam03] N. Mamoulis, “Efficient processing of joins on set-valued attributes,” in *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, 2003, pp. 157–168.

[Mat09] C. Mathis, “Storing, indexing, and querying xml documents in native xml database systems,” Ph.D. dissertation, Technische Universität Kaiserslautern, 2009.

[ME97] A. E. Monge and C. Elkan, “An efficient domain-independent algorithm for detecting approximately duplicate database records,” in *Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD’97)*, 1997, pp. 0–.

[MLK$^+$05] R. Murthy, Z. H. Liu, M. Krishnaprasad, S. Chandrasekar, A.-T. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, and V. Krishnamurthy, “Towards an enterprise xml architecture,” in *Proc. SIGMOD Conf.*, 2005, pp. 953–957.

[MML07] M. M. Moro, S. Malaika, and L. Lim, “Preserving xml queries during schema evolution,” in *Proc. of the 16th Intl. Conf. on World Wide Web (WWW 2007)*, 2007, pp. 1341–1342.

[MMR05] F. Mandreoli, R. Martoglia, and E. Ronchetti, “Versatile structural disambiguation for semantic-aware applications,” in *Proc. of the 2005 ACM*

CIKM Intl. Conf. on Information and Knowledge Management (CIKM 2005), 2005, pp. 209–216.

[MN08] F. McSherry and M. Najork, "Computing information retrieval performance measures efficiently in the presence of tied scores," in *Proc. of the 30th European Conference on IR Research (ECIR 2008)*, 2008, pp. 414–421.

[MNU00] A. McCallum, K. Nigam, and L. H. Ungar, "Efficient clustering of high-dimensional data sets with application to reference matching," in *Proc. of the 6th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (KDD 2000)*, 2000, pp. 169–178.

[MRLG93] D. L. Medin and D. G. Robert L. Goldstone, "Respects for similarity," *Psychological Review*, vol. 100, pp. 254–278, 1993.

[MS99] T. Milo and D. Suciu, "Index structures for path expressions," in *Proc. of the 7th International Conference on Database Theory (ICDT'99)*, 1999, pp. 277–295.

[MSC06] D. Milano, M. Scannapieco, and T. Catarci, "Structure aware xml object identification," 2006.

[MW04] A. McCallum and B. Wellner, "Conditional models of identity uncertainty with application to noun coreference," in *Proc. of the 17th Conf. on Neural Information Processing Systems (NIPS 2004)*, 2004, pp. 905–912.

[Nav01] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[Nav09] R. Navigli, "Word sense disambiguation: A survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. 2, 2009.

[NJ02] A. Nierman and H. V. Jagadish, "Evaluating structural similarity in xml documents," in *Proc. of the 5th Intl. Workshop on the Web and Databases (WebDB 2002)*, 2002, pp. 61–66.

[NKAJ59] H. Newcombe, J. Kennedy, S. Axford, and A. James, "Automatic linkage of vital records," *Science*, vol. 130, no. 3381, pp. 954–959, 1959.

[OC03] P. Ogilvie and J. P. Callan, "Combining document representations for known-item search," in *Proc. SIGIR Conf.*, 2003, pp. 143–150.

[OOP$^+$04] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury, "Ordpaths: Insert-friendly xml node labels," 2004, pp. 903–908.

[PC98] J. M. Ponte and W. B. Croft, "A language modeling approach to information retrieval," in *SIGIR '98: Proc. of the 21st Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR'98)*, 1998, pp. 275–281.

[Pea88] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers, 1988.

[PMAJ01] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran, "Block oriented processing of relational database operations in modern computer architectures," in *Proc. of the 17th Intl. Conf. on Data Engineering*, 2001, pp. 567–574.

[PMM$^+$02] H. Pasula, B. Marthi, B. Milch, S. J. Russell, and I. Shpitser, "Identity uncertainty and citation matching," in *Proc. of the 15th Annual Conf. on Neural Information Processing Systems (NIPS 2002)*, 2002, pp. 1401–1408.

[Por80] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[PWN06] S. Puhlmann, M. Weis, and F. Naumann, "Xml duplicate detection using sorted neighborhoods," in *Proc. of the 10th Intl. Conf. on Extending Database Technology (EDBT 2006)*, 2006, pp. 773–791.

[Pyl99] D. Pyle, *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, 1999.

[RB01] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[RDM04] E. Rahm, H. H. Do, and S. Massmann, "Matching large xml schemas," *SIGMOD Record*, vol. 33, no. 4, pp. 26–31, 2004.

[RH07] L. A. Ribeiro and T. Härder, "Embedding similarity joins into native xml databases," in *Anais do XXII Simpósio Brasileiro de Banco de Dados*, 2007, pp. 285–299.

[RH08a] L. A. Ribeiro and T. Härder, "Evaluating performance and quality of xml-based similarity joins," in *Proc. of the 12th East European Conference on Advances in Databases and Information Systems (ADBIS 2008)*, 2008, pp. 246–261.

[RH08b] L. A. Ribeiro and T. Härder, "Similarity matching in web-based data management applications," *Datenbank-Spektrum*, vol. 26, 2008.

[RH09] L. A. Ribeiro and T. Härder, "Efficient set similarity joins using min-prefixes," in *Proc. of the 13th East European Conf. on Advances in Databases and Information Systems (ADBIS 2009)*, 2009, pp. 88–102.

[RHMGM09] D. Ramage, P. Heymann, C. D. Manning, and H. Garcia-Molina, "Clustering the tagged web," in *Proc. of the 2sd Intl. Conf. on Web Search and Web Data Mining (WSDM 2009)*, 2009, pp. 54–63.

[RHP09] L. A. Ribeiro, T. Härder, and F. S. Pimenta, “A cluster-based approach to xml similarity joins,” in *International Database Engineering and Applications Symposium (IDEAS 2009)*, 2009, pp. 182–193.

[Ric08] M. M. Richter, “Similarity,” in *Case-Based Reasoning on Images and Signals*, 2008, pp. 25–90.

[RN03] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[Rus18] R. C. Russel, “Russel index,” http://patft.uspto.gov/netahtml/srchnum.htm, 1918.

[RW94] S. E. Robertson and S. Walker, “Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval,” in *Proc. of the 17th Annual Intl. ACM-SIGIR Conf. on Research and Development in Information Retrieval (Special Issue of the SIGIR Forum)*, 1994, pp. 232–241.

[Rys05] M. Rys, “Xml and relational database management systems: Inside microsoft sql server 2005,” in *Proc. SIGMOD Conf.*, 2005, pp. 958–962.

[RZT04] S. E. Robertson, H. Zaragoza, and M. J. Taylor, “Simple bm25 extension to multiple weighted fields,” in *Proc. of the 2004 ACM CIKM Intl. Conf. on Information and Knowledge Management*, 2004, pp. 42–49.

[SAA10] Y. N. Silva, W. G. Aref, and M. H. Ali, “The similarity join database operator,” in *Proc. of the 26th Intl. Conf. on Data Engineering (ICDE 2010)*, 2010, pp. 892–903.

[Sar08] S. Sarawagi, “Information extraction,” *Foundations and Trends in Databases*, vol. 1, no. 3, pp. 261–377, 2008.

[SB02] S. Sarawagi and A. Bhamidipaty, “Interactive deduplication using active learning,” in *Proc. of the 8th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD 2002)*, 2002, pp. 269–278.

[SD06] P. Singla and P. Domingos, “Entity resolution with markov logic,” in *Proc. of the 6th IEEE Intl. Conf. on Data Mining (ICDM 2006)*, 2006, pp. 572–582.

[Seb02] F. Sebastiani, “Machine learning in automated text categorization,” *ACM Computing Surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[Sed05] E. Sedlar, “Managing structure in bits & pieces: the killer use case for xml,” in *Proc. SIGMOD Conf.*, 2005, pp. 818–821.

[SK03] S. Sarawagi and A. Kirpal, “Scaling up the alias duplicate elimination system: A demonstration,” in *Proc. of the 19th Intl. Conf. on Data Engineering (ICDE 2003)*, 2003, pp. 783–785.

[SK04] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2004)*, 2004, pp. 743–754.

[SM83] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc. New York, 1983.

[sql03] "Information organization for standardization (iso). information technology— database languages— sql—part 14: Xml-related specifications (sql/xml)," 2003.

[SSB05] E. Spertus, M. Sahami, and O. Buyukkokten, "Evaluating similarity measures: a large-scale study in the orkut social network," in *Proc. of the 11th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2005, pp. 678–684.

[Ste07] B. Stein, "Principles of hash-based text retrieval," 2007, pp. 527–534.

[STZ$^+$99] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton, "Relational databases for querying xml documents: Limitations and opportunities," in *Proc. of 25th Intl. Conf. on Very Large Data Bases (VLDB'99)*, 1999, pp. 302–314.

[Tai79] K.-C. Tai, "The tree-to-tree correction problem," *Journal of the ACM*, vol. 26, no. 3, pp. 422–433, 1979.

[TBM$^+$08] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum, "Topx: efficient and versatile top- query processing for semistructured data," *The VLDB Journal*, vol. 17, no. 1, pp. 81–115, 2008.

[TF95] H. R. Turtle and J. Flood, "Query evaluation: Strategies and optimizations," *Information Processing Management*, vol. 31, no. 6, pp. 831–850, 1995.

[TKM02] S. Tejada, C. A. Knoblock, and S. Minton, "Learning domain-independent string transformation weights for high accuracy object identification," in *Proc. of the 8th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2002, pp. 350–359.

[TNB08] T. Tran, R. Nayak, and P. Bruza, "Combining structure and content similarities for xml document clustering," in *Proc. of the 7th Australasian Australasian Data Mining Conference (AusDM 2008)*, 2008, pp. 219–226.

[TS04] A. Trotman and B. Sigurbjörnsson, "Narrowed extended xpath i (nexi)," in *Proc. of the 3rd Intl. Workshop of the Initiative for the Evaluation of XML Retrieval (INEX 2004)*, 2004, pp. 16–40.

[TSK06] P.-N. Tan, M. Steinbach, and V. Kumar, "Introduction to data mining." Addison-Wesley, 2006.

[TSP08] M. Theobald, J. Siddharth, and A. Paepcke, "Spotsigs: Robust and efficient near duplicate detection in large web collections," in *Proc. of the 31st Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR 2008)*, 2008, pp. 563–570.

[TSW03] M. Theobald, R. Schenkel, and G. Weikum, "Exploiting structure, annotation, and ontological knowledge for automatic classification of xml data," in *Proc. WebDB Conf.*, 2003, pp. 1–6.

[TSW05] M. Theobald, R. Schenkel, and G. Weikum, "An efficient and versatile query engine for topx search," in *Proc. of the 31st Intl. Conf. on Very Large Data Bases (VLDB 2005)*, 2005, pp. 625–636.

[TW01] A. Theobald and G. Weikum, "Adding relevance to xml," 2001, pp. 105–124.

[Ukk92] E. Ukkonen, "Approximate string matching with q-grams and maximal matches," *Theoretical Computer Science*, vol. 92, no. 1, pp. 191–211, 1992.

[VCÖ$^+$07] Z. Vagena, L. S. Colby, F. Özcan, A. Balmin, and Q. Li, "On the effectiveness of flexible querying heuristics for xml data," in *Proc. of the 5th Intl. XML Database Symposium (XSym 2007)*, 2007, pp. 77–91.

[VHdSdM07] A. R. Vinson, C. A. Heuser, A. S. da Silva, and E. S. de Moura, "An approach to xml path matching," in *9th ACM International Workshop on Web Information and Data Management (WIDM 2007)*, 2007, pp. 17–24.

[Via01] V. Vianu, "A web odyssey: From codd to xml," in *Proc. of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2001)*, 2001, pp. 1–15.

[Wei07] G. Weikum, "Db&ir: both sides now," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2007)*, 2007, pp. 25–30.

[Win88] W. E. Winkler, "Using the em algorithm for weight computation in the fellegi-sunter model of record linkage," in *Proc. of the Section on Survey Research Methods of the American Statistical Association*, 1988, pp. 667–671.

[Win06] W. Winkler, "Overview of record linkage and current research directions," Statistical Research Division, U.S. Bureau of the Census, Tech. Rep., 2006.

[WM89] Y. R. Wang and S. E. Madnick, "The inter-database instance identification problem in integrating autonomous systems," 1989, pp. 46–55.

[WM07] M. Weis and I. Manolescu, “Declarative xml data cleaning with xclean,” in *Proc. of the 19th Intl. Conf. on Advanced Information Systems Engineering (CAISE 2007)*, 2007, pp. 96–110.

[WMB99] I. H. Witten, A. Moffat, and T. C. Bell, *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[WMK$^+$09] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina, “Entity resolution with iterative blocking,” in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2009)*, 2009, pp. 219–232.

[WN05] M. Weis and F. Naumann, “Dogmatix tracks down duplicates in xml,” in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2005)*, 2005, pp. 431–442.

[WSB98] R. Weber, H.-J. Schek, and S. Blott, “A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces,” in *Proc. of 24th Intl. Conf. on Very Large Data Bases (VLDB’98)*, 1998, pp. 194–205.

[xml09] “Xml schema,” 2009, http://www.w3.org/XML/Schema.

[XP05] Y. Xu and Y. Papakonstantinou, “Efficient keyword search for smallest lcas in xml databases,” 2005, pp. 537–538.

[xtc] “The xtc project: Native xml data management,” http://wwwlgis.informatik.uni-kl.de/cms/dbis/projects/xtc/.

[XWL08] C. Xiao, W. Wang, and X. Lin, “Ed-join: An efficient algorithm for similarity joins with edit distance constraints,” *Proc. of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 933–944, 2008.

[XWLS09] C. Xiao, W. Wang, X. Lin, and H. Shang, “Top-k set similarity joins,” in *Proc. of the 25th Intl. Conf. on Data Engineering (ICDE 2009)*, 2009, pp. 916–927.

[XWLY08] C. Xiao, W. Wang, X. Lin, and J. X. Yu, “Efficient similarity joins for near duplicate detection,” in *Proc. of the 17th Intl. Conf. on World Wide Web (WWW 2008)*, 2008, pp. 131–140.

[YASU01] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura, “Xrel: a path-based approach to storage and retrieval of xml documents using relational databases,” *ACM Transactions on Internet Technology*, vol. 1, no. 1, pp. 110–141, 2001.

[YJF98] B.-K. Yi, H. V. Jagadish, and C. Faloutsos, “Efficient retrieval of similar time sequences under time warping,” in *Proc. of the 14th Intl. Conf. on Data Engineering (ICDE’98)*, 1998, pp. 201–208.

[YKT05] R. Yang, P. Kalnis, and A. K. H. Tung, “Similarity evaluation on tree-structured data,” in *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2005)*. ACM, 2005, pp. 754–765.

[YP97] Y. Yang and J. O. Pedersen, “A comparative study on feature selection in text categorization,” in *Proc. of the 14th Intl. Conf. on Machine Learning (ICML 1997)*, 1997, pp. 412–420.

[YQJ02] C. Yu, H. Qi, and H. V. Jagadish, “Integration of ir into an xml database,” in *Proc. of the 1st Workshop of the INitiative for the Evaluation of XML Retrieval (INEX)*, 2002, pp. 162–169.

[YSLH03] K.-P. Yee, K. Swearingen, K. Li, and M. A. Hearst, “Faceted metadata for image search and browsing,” in *Proc. CHI Conf.*, 2003, pp. 401–408.

[Zak02] M. J. Zaki, “Efficiently mining frequent trees in a forest,” in *Proc. of the 8th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2002, pp. 71–80.

[ZB06] J. Zobel and Y. Bernstein, “The case of the duplicate documents measurement, search, and science,” in *Proc. of Frontiers of WWW Research and Development (APWeb 2006)*, 2006, pp. 26–39.

[ZL77] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, 1977.

[ZM93] J. Ziv and N. Merhav, “A measure of relative entropy between individual sequences with application to universal classification,” *IEEE Transactions on Information Theory*, vol. 39, no. 4, pp. 1270–1279, 1993.

[ZM98] J. Zobel and A. Moffat, “Exploring the similarity space,” *SIGIR Forum*, vol. 32, no. 1, pp. 18–34, 1998.

[ZS89] K. Zhang and D. Shasha, “Simple fast algorithms for the editing distance between trees and related problems,” *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[ZSS92] K. Zhang, R. Statman, and D. Shasha, “On the editing distance between unordered labeled trees,” *Information Processing Letters (IPL)*, vol. 42, no. 3, pp. 133–139, 1992.

Curriculum Vitae

Leonardo Andrade Ribeiro

Personal Information

Date of Birth:	October 13, 1976
Place of Birth:	Quirinópolis, Brazil
Nationality:	Brazilian
Marital Status:	Single

Contact Information

University of Kaiserslautern,
Dept. of Computer Science, AG DBIS
P.O. Box 3049
D-67653 Kaiserslautern, Germany
Phone: +49-631-205-2159
Fax: +49-631-205-3299
Room: 36/320

Education

1986–1989
Secondary School, Colégio Estadual Independência, Quirinópolis, Brazil.

1990–1992
Secondary School, Colégio Educacional de Quirinópolis (CEQ), Quirinópolis, Brazil.

1994–2000

Bachelor of Computer Science, Universidade Federal de Goiás (UFG), Gôiania, Brazil.

2000–2002

Master of Sciences
Dissertation: A gateway for distributed medical image sources using CORBA for integration of teleradiology services.
Supervisor: Prof. Dr. Aldo von Wangenheim.
Universidade Federal de Santa Catarina (UFSC)
Santa Catarina, Brazil.

Research Experience

2002

Research Assistant on the Germany/Brazilian Cyclops Project. Research field: Heterogeneous medical databases.

October 2002–November 2002

Guest researcher at the University of Kaiserslautern, Germany. Research activities: Healthcare decision support tools in joint work with the research team of Prof. Dr. Michael M. Richter.

January 2003–June 2003

Research Assistant on the I2TV Project. Research field: Framework for digital interactive television.

July 2003–June 2004

Research Assistant on the Centro Integrado Multidisciplinar de Pesquisas em Bioinformtica de Santa Catarina Project. Research field: Parallel computing for bioinformatics.

Since April 2005

Scientific staff member of Database and Information Systems Group at the University of Kaiserslautern, Germany.