

Accelerated Schema Matching Using B-trees

Duchateau Fabien

MSc. in Computing

University of Montpellier II - LIRMM/INFO

2006

Directors of research : Dr Zohra Bellahsene and Dr Mark Roantree.

University of Montpellier II - LIRMM/INFO
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Duchateau Fabien

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

Name of Chairperson

Reading Committee:

Name of Committee member

Name of Committee member

etc

Date: _____

In presenting this thesis in partial fulfillment of the requirements for a master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purpose or by any means shall not be allowed without my written permission.

Signature_____

Date_____

University of Montpellier II - LIRMM/INFO

Abstract

Accelerated Schema Matching Using B-trees

Duchateau Fabien

Chair of the Supervisory Committee:

Professor Name of Chairperson

Chair's department

Integration process is the problem of combining information located at different sources and providing the user with a unified view of these data. A user query against this unique interface must be split and rewritten for each data sources containing answers to this query, and the results have to be aggregated to give the user a comprehensive but complete answer. With the emergence of the Internet and its possibilities to share thousands of information sources, many research domains like environment, health, e-commerce, etc. need this schema integration process. Once it is correctly done, the advantages are numerous, including fast decision-taking or new knowledge generation. As the manual schema matching is not reliable and a loss of time, many algorithms have been designed to find mappings between schemas. Most of them offer a good quality matching but do not provide a fast matching with a large number of schemas. We show in this paper that using a B-tree structure improve significantly performances namely in large scale scenario and provide acceptable quality of mappings.

TABLE OF CONTENTS

List of Figures	ii
Chapter 1: Introduction	1
1.1 Background	1
1.2 Motivation	2
1.3 Issues	3
1.4 Summary	4
Chapter 2: Related work	5
2.1 COMA++	7
2.2 Charlie	9
2.3 Cupid	11
2.4 Glue	13
2.5 Clio	15
Chapter 3: Adopting a Btree Approach to Schema Matching	17
3.1 Using B-trees	18
3.2 Overview of Btree Match	19
3.3 Detailed discussion of the process	21
3.4 Example	28
3.5 Support for dynamic processing	30
Chapter 4: Results and experiments	34
4.1 Parameters in Btree Match	34
4.2 Comparisons	35
Chapter 5: Conclusions	39
Bibliography	41

LIST OF FIGURES

Figure Number	Page
3.1 An example of B-tree of <i>order</i> 5.	18
3.2 Examples of XML Schemas.	20
3.3 The paths of each node used as indexes in the B-tree.	20
3.4 Architecture of Btree Match.	24
3.5 Screenshot of Btree Match.	27
3.6 The B-tree after building it from <i>SyncSalesOrder.xsd</i>	28
3.7 The B-tree after integrating <i>UpdateSalesOrder.xsd</i>	29
4.1 Varying <i>ORDER</i> parameter has no significant impact.	34
4.2 <i>NB_ANCESTORS_TO_GO</i> parameter can provide better quality matching.	35
4.3 Matching comparison with Charlie on the number of schemas.	36
4.4 Matching comparison with Charlie on the size of schemas.	37
4.5 Matching comparison with Coma++.	38

ACKNOWLEDGMENTS

I would like to thank firstly my research mentor, Dr Zohra Bellahsene, to have enabled me to work in collaboration with ISG team at Dublin City University, and for her guidance and encouragement during those five months.

I also thank my research mentor in Dublin, Dr Mark Roantree, for his welcoming among his team, but also for his precious advice for the writing of the paper and his expert guidance.

I would like to thank Colm and Cian, members of Mark's team, who helped me much once I arrived in DCU. They participated in my training, especially the use of Latex, and they were always here to answer my questions when needed.

Finally I wish to express sincere appreciation to both universities, Universite Montpellier II and Dublin City University, where I had the opportunity to use the material and to learn new knowledge.

Chapter 1

INTRODUCTION

1.1 Background

Inside organisations, there are many autonomous systems storing data. In an external environment, it is also the case and users and systems need to query multiple sources simultaneously. That is why in late 80's, federated databases emerged, in which data sources could be geographically decentralized but connected though a network to offer end-users the possibility to query the whole system with an uniform interface hiding the distribution and the heterogeneity of the data. Even if the benefits are numerous and immediate, these federated databases had some drawbacks : designing a global schema for a whole organisation could be a really hard work, especially the cleaning process. As a consequence, updates could not be easily managed. And in case of a crash in the global schema computer, the whole system is out of service until the problems are fixed.

Besides information were stored on different supports : they could be found in simple text documents, in databases, or more recently by using description languages like XML¹ and RDF². Those languages, especially XML, enable to solve the difficulty of heterogeneous sources. So this problem is nowadays often reduced to XML sources as many systems provide XML querying multiple interfaces. Indeed XML is one of the most used language with both human and machine-readable format and an easy possibility to share information across different systems. XML, by describing, structuring and containing data, logically spread and is now widely used when dealing with large quantities of data that need to be

¹eXtensible Markup Language

²Resource Description Framework

shared and exploited, among which integration process.

1.2 Motivation

Integration process is the problem of combining information located at different sources and providing the user with a unified view of these data. A user query against this unique interface must be split and rewritten for each data sources containing answers to this query, and the results have to be aggregated to give the user a comprehensive but complete answer. With the emergence of the Internet and its possibilities to share thousands of information sources, many research domains like environment, health, e-commerce, etc. need this schema integration process. Integration of web data sources or datawarehouse loading are some examples illustrating the importance of this step. Once it is correctly done, the advantages are numerous, including fast decision-taking or new knowledge generation.

However researchers quickly had to face some difficulties in the integration process : data from multiple sources are often formatted in incompatible ways, meaning that elements name are probably different from one source to another, and even worse, represented using incompatible assumptions, with some of them totally implicit. For instance, information in the first source could be measured in meters whereas the unit of another source could be in feet. In case of missing data, the first source may just fill in with a default value while the other source will let a blank field. Furthermore the data sources are often stored in different databases or schemas. Thus data integration brought many problems and the one we address in this report is schema matching.

Schema matching consists in identifying semantic correspondences between elements of different schemas. Schema matching is currently a manual or semi-automatic process, thus a steady and error-prone step resulting in a loss of time and many mistakes. As the number of data sources increase daily, especially on the Internet, it is necessary to automatize this process. Two features are important to obtain a good matcher tool : its performances must be efficient, in other words the algorithm should execute the matching in a reasonable time,

and secondly the matching quality must be good enough to avoid too much human checking.

1.3 Issues

Data schemas have nowadays many formats : XSD³, RDF, OWL⁴ or databases schemas are the most used. They all have some good points or specific functionalities so people choose the one best appropriate for their needs. That is why there are a lot of schema formats, and why schema matching is so crucial. Many techniques were developed to solve the matching problem : some of them exploit names and types properties thanks to natural language processing algorithms or dictionaries, some focuses on the schema structure to extract similarity subsets while others are based on mathematical formulas. Currently none of these techniques proved to be better than others : worse, the best matching applications gives better results when they use different techniques, which in the other hand decrease performances. Therefore there is an important need to improve performances for matching process.

Another aspect is the dynamicity of the data sources : as illustrated by peer-to-peer networks and the increasing number of laptops, information can be modified, added or it could simply vanish. Thus it becomes very common that information is continuously updated and these changes need to be reflected in the mappings. Indeed it is necessary - if not compulsory - to keep up-to-date relationships to maintain a reliable network of data sources that is efficiently exploited. But again, this context is viable only if the matching is a fast process.

Even if many matchers can be found, like COMA++, Cupid or Glue, which provide a good matching quality, their main drawback concerns the speed of the algorithms execution. And especially if we consider dynamic aspect, then the matching process cannot be executed during several hours. The main idea of this thesis is to propose a solution that

³XML Schema Description

⁴Web Ontology Language

both provide good performances and offers a good matching quality. Further perspectives could then be shown especially for dynamic aspect.

1.4 Summary

We have seen that data integration is a well-known problem for several decades, but it still needs to be solved and adapted because of the growing quantity of information and techniques due to the great heterogeneity of the data sources. The XML standard follows this direction by enabling sharing the data. Schema matching is a crucial step in the data integration process since it concerns the discovery of relationships between elements of several schemas. Much work has been done on schema matching but this problem is so complex that it requires more studies to obtain reliable tools which automatizes this process. Besides new challenges appear with the performance aspect namely in the large scale context. Furthermore, it is necessary to deal with the evolution the evolution and maintenance aspect of the discovered mappings.

This report is structured as follows: in **Chapter 2**, we discuss similar projects; in **Chapter 3**, an outline of our B-tree match method is described with algorithms and illustrated by an example; in **Chapter 4**, we provide experiments of our approach and discuss their results regarding some other related work; and in **Chapter 5**, we offer some conclusions and future work.

Chapter 2

RELATED WORK

In this section, we review some approaches and the related tools that are used for schema matching. First we give some formal definitions.

Definition 1 : A schema is a labeled unordered tree $S = (V_S, E_S, r_S, label)$ with :

- V_S is a set of nodes;
- r_S is the root node;
- $E_S \subseteq V_S \times V_S$ is a set of edges;
- $label V_S \rightarrow \Lambda$ where Λ is a countable set of labels.

Definition 2 : A mapping is a relationship between nodes of different schemas, but the type of this relationship is not defined. It can include synonyms, equality, hyperonyms, etc.

Definition 3 : Let V be the domain of schema nodes, the semantic distance is a value $V \times V \rightarrow \mathfrak{R}$ defined for two nodes and which indicates the similarity between these nodes according to the match criteria. A zero value means a total similarity whereas an infinite value stands for completely different concepts.

There are many techniques used to find a match between elements of different schemas. The survey presented in [5] gives an overview of those techniques. To sum up, they can be sorted in different categories :

- **Schema or instance**, the first one deals only with metadata found in the schemas whereas the second uses machine learning or statistics directly on the data or metadata.

- **Individual or hybrid**, meaning that the application applies only one algorithm for the matching process (individual) or it can use a combination of different algorithms (hybrid).
- **Linguistic**, this is commonly used since it concerns exploitation of the names and descriptions. It implies many steps to process the matching, for example tokenization (*PostCode_Address* is derived from *PostCode* and *Address*), lemmatization (*Firstnames* gives *Firstname*), elimination (*IsRelatedTo* is abbreviated in *Related*).
- **Auxiliary**, often associated with linguistic since it uses dictionaries or thesaurus to find synonyms, meanings, and other relationships with elements names.
- **Constraint based**, which consists in exploiting information found in the schemas, like value range, data types, unicity and other attributes to find mappings.
- **Cardinality**, this represents the matching cardinality, e.g. the number of elements to be mapped. For instance, $1:n$ means that the algorithm try to find, for each element of the source schema, one to n matching elements in the target schema.
- **Element or group**, dealing with the structure. The matcher can use either one element or a group of elements that appear together in a schema. For example, *Address*, only one element, could be found as a group of elements - namely *Street*, *PostCode*, *City* - in another schema.

Note that most of the matchers use some linguistic techniques, and that the best quality matchers work with several of those methods then combines theirs results to obtain the most plausible mappings. In the next sub section, we will present an overview of some of those tools, with their advantages and their drawbacks.

2.1 COMA++

As described in [3] COMA++ is a hybrid matching tool, it gathers many independent matching algorithms. Thus it is quite difficult to classify COMA++ in one of the categories. At least it works with schemas and supports many formats including RDF and XSD¹. Different strategies could be tested that offer variables results : the user needs to try them by choosing which algorithms, and how to combine their results. For instance, the reuse-oriented matching enables to use last mappings found or the fragment-based matching allows to decompose large match problems into smaller subproblems. When loading a schema, COMA++ transforms it into a rooted directed acyclic graph and its elements are represented by graph nodes linked by different relationships like containment or referential. This matcher enables only two schemas to be matched but the mappings found can be used then to accelerate other matching. COMA++ does not use any dictionary or ontology, but has a list of synonyms that can be extended by the user. All schemas and discovered mappings are stored in a repository implemented with MySQL.

More precisely, it works as follows : first the two schemas to be matched are loaded from the repository. This is a slow step since the schemas stored in a generic data model are transformed into a directed graph. Then the user needs to select in the *matcher library* the match algorithms he wants to use. For each selected match algorithm, each element from the source schema is attributed a threshold value between 0 (no similarity) and 1 (total similarity) with each element of the target schema, resulting in a *cube of similarity values*. Even if COMA++ focuses on 1:1 relationships, the match algorithms often select several possible mappings for one element. The final step consists in combining the similarity values given by each matcher algorithm : either the maximum value or the average value is kept for every mapping found, then another selection strategy with a threshold value applies to determine the best mappings in case of several possibilities. Finally, COMA++ displays all the mappings possibilities and the user has just to check and validate the correct mappings.

¹XML Schemas Description

The advantage of COMA++ is certainly its good matching quality, and the possibility to re-use mappings. It supports many formats, including ontologies. The graphical interface has been smartly thought to provide ease of use. During the match process or in the end, the user has the final decision to choose the appropriate mappings since COMA++ has done most of the jobs by selecting those possibles. Besides new matching algorithms can be added and the list of synonyms can be completed, thus offering some perspectives for specific field areas. It is also a good platform to evaluate and compare new matching algorithms.

However the weak point of COMA++ is probably the time spent both for adding the files in the repository and to match schemas. In a large scale context, spending several minutes with those operations can entail performance degradation and the other drawback is that it does not support the matching of many schemas directly. The parameters tuning and the choice of the matchers and strategies to apply may also appear fastidious for the user, and he would probably have to try several configurations before finding an optimal matching. Although the taxonomy matcher allows to use an ontology as an intermediate link between two schemas, a general dictionary could improve significantly the system, even if this would imply slower performances. It could obviously be painful for the user to complete the list of synonyms for large specific domains like biology or physics. Finally COMA++ needs a database (currently MySQL) to store the schemas and mappings so it depends on this software.

2.2 Charlie

We then study Charlie, a schema matching method that has been developed at LIRMM, Montpellier and presented in [13]. It also aims at improving time spent on matching for both XML documents and XML schemas, and has been specifically designed for the XPeer architecture : this mediation system enables to hide the distribution, heterogeneity and localization of data sources when querying. Charlie's algorithm consists in parsing each of the documents and storing its information (path, name, parent, ... of the nodes) in a tree structure, then it matches the trees : for each element, it searches a match in other trees by exploring level by level, but starts its search at the last matched ancestor when possible, and extends it to the children nodes first until a certain depth is reached. If needed it can also backward and find a mapping in the ancestors of the last mapping found and their siblings. For each met node, Charlie applies a semantic distance, a value between 0 ('perfect' matching) and 1 (no matching). The algorithm does not match only with the node which has the smaller semantic distance : indeed there is a threshold *maxdist*, defined by the user, below which the semantic distance must be. Finally a mediated document is built and the mappings found can be stored in a text file.

Charlie can take as inputs both XML documents and XML schemas and matches fast a large number of schemas. Its 2 parameters, *maxdepth* and *maxbacktrack*, enables to prioritize either speed or matching quality. The last parameter, *maxdist*, is a good idea that allows also to adjust the matching quality. However those mappings should be tuned automatically by the system instead of needing human intervention. Charlie was designed for large scale context, especially for the XPeer architecture. Now it just need to be integrated in a peer-to-peer architecture to be totally efficient by exploiting the domains in which each peer belongs.

However, Charlie's performance are not always reliable : the mediated schema produced is sometimes wrong and forgets some elements. Besides it is based on DOM parser, which is very efficient with small documents, but is slower than SAX parser for example when dealing

large files. Finally, Charlie uses the semantic distance to compare nodes, but this notion needs to be defined further or extended using an ontology or a dictionary. The mappings are just written in a text file while they should either be stored in memory or directly used to rewrite a query. Finally the first schema chosen by Charlie has some influence on the performances, so this should be fixed so that it chooses always the best adapted schema.

2.3 *Cupid*

Cupid is a prototype tool that was designed to be generic in [4]. The main idea is that a robust matching method needs many matching techniques to provide the best results. Cupid works only with schemas and is both group and element based. Although it mainly uses linguistic, it also exploits some constraints. Its algorithm is divided in two phases : in the first one it computes some coefficient to find similarities between elements names, data types, and deduces some mappings from them. Then they browse the structure to find more mappings. For instance, if one element e_1 has some children matching the children of element e_2 , and e_1 's parent matching e_2 's parent, there are lots of chances that e_1 could match e_2 . Cupid does not consider input schemas as trees, since in the real world there are many relationships of containment, aggregation and other constraints that lead to non-tree schemas. Thus schemas are merely seen as graphs that can handle complex relationships like inheritance or foreign key integrity constraint.

In the first phase, the linguistic process, is divided in three steps :

- **normalization**, which consists in finding similar elements that are yet spelled differently by transforming their names into tokens, removing punctuation (tokenization). Then another process is necessary to identify abbreviated words (expansion), and finally articles and prepositions need to be discarded (elimination). All this work could not be done without a thesaurus, which sometimes needs to be completed by specific domain references.
- **categorization**, to sort out elements. Indeed schema elements belong to one or more categories, according to their data types, names, etc. This step enables only not to compare elements that do not belong to the same category in the next step.
- **comparison**, with a thesaurus. Synonyms and hypernymy are found at this step. Substring search is also done and the results of this step is a linguistic coefficient, between 0 and 1, for elements of different schemas.

The second phase intends to generate structure coefficient between elements of different schemas, by using the linguistic coefficient already calculated. The TreeMatch algorithm searches with preorder traversal to find mappings in the leaves first, then searches again to find mappings in non-leaves nodes. Finally a set of mappings is generated

The advantages of Cupid are numerous. It offers a good matching quality, especially thanks to structural traversal that complete linguistic process. Some options enables to re-use found mappings, that may have been checked and corrected by the user. In that case, the result of the new matching process is often better since Cupid marks user-given mappings as reliables.

However Cupid suffers from several weak points : the schemas need to be parsed at least three times. This is a tedious job, which could not be done in a large scale, but necessary as the matching quality decreases without the structure traversal. As its authors explain, some mappings cannot be discovered because the parameters need to be tuned by an expert. Auto-tuning these parameters is required so that the application might be used by non-IT users. More, the good matching quality really decreases without an appropriate thesaurus, mostly in specific domain area.

2.4 *Glue*

Glue is above all a semantic matcher, so it is better appropriate to work with ontologies, even if it could be used for schemas too since they could be viewed like ontologies with restrictive relationships. Glue aims at finding relationships between two taxonomies. Those semantic mappings are discovered thanks to machine learning techniques. [8] gives many details about the architecture of Glue. It describes the three components, namely the *distributor estimator*, the *similarity estimator* and the *relaxation labeler*. The first one takes two ontologies as input, and it will calculate some joint distributions values between each pair of elements from both taxonomies. Those values feeds the *similarity estimator* that computes similarity measures chosen by the user. With the name learner and the content learner, there is also a meta-learner which estimates how it trusts the other learners² and then combine their values. The results of this second component is a matrix of similarity values between concepts of the two ontologies. Finally the *relaxation labeler* searches, from the similarity matrix, the best mappings configuration which satisfies a set of constraints, some of them being domain-dependent and others domain-independent. The relaxation label idea exploits the influence of a node on its neighboring nodes. A label is assigned to the node, then this label may be changed according to the neighboring nodes, and the process stops when the label does not change anymore. This method has already been applied with success in several domains, including computer vision or natural language processing.

When studying the experiments, we notice a very good matching quality reaching sometimes 97%. Matching ontologies may be very useful since using dictionaries like Wordnet might result in poor accuracy due to the non-domain specific and large information it contains. So it could be far better to match several ontologies from the same domain and then use this new taxonomy to match many schemas than using a general dictionary. Besides the ontologies used for the experiments contains many nodes, around 200, which is more realistic than schemas with only a few nodes as we can see in some other tests. The good

²This trust parameter is set manually for the moment but designers showed it could be computed automatically.

results are probably due to the multi-strategy learner approach and the meta-learner which are good ideas since each learner exploits different information, for example frequencies of words, value formats, etc. And new learners may easily be added to suit new needs. By computing joint distributions, Glue can then calculate many similarity measure, and could still add new ones if required. These notions are based on mathematics and probabilities, and thus have good foundations.

Glue can match only two ontologies at the same time. Moreover, data instances are much used by Glue matching algorithms while it is not obvious that ontologies and schemas have these instances. They could just describe the data without giving some samples. We wonder what results would give Glue if no data instances can be found in the taxonomies, even if the authors proves that a moderate number of data instances is necessary to obtain a good matching quality. The domain-dependent constraints must be set up by an expert because they are modelised as formulas. The learners need some training examples, but what happens if the users cannot provide them. And the results of Glue strongly depends on the number and the quality of these training examples. Performances are not described so we do not know how long it takes to perform the matching.

2.5 *Clio*

Developed at IBM and presented in [9], *Clio* is an information integration tool that offers a user-friendly GUI for matching. Its match cardinality is $n:1$, meaning that one or more schemas are mapped to a target schema. Those schemas are mainly relational databases like DB2 or Oracle, but also XML. *Clio* goes further by exploiting the query notion. Indeed a set of queries is generated, enabling the data from the source schemas to be integrated into the target schema. This is very useful to fill in a datawarehouse.

Like COMA++, the user can intervene to define mappings with the GUI, and *Clio* generates the adequate queries for those mappings. But *Clio* also assists the user by suggesting the mappings. Its architecture and algorithm are deeply described in [9]. To sum up, *Clio* has three components, each corresponding to one step in the matching process : *schema engine*, *correspondence engine* and *mapping engine*. The first one is in charge of loading a schema from a file or a database wrapper. It may add some constraints to the schema, for instance foreign keys. However the user is always asked to check the validity of these added information thanks to a comprehensive graphical user interface that enables them both to see a view of the schema and an example of data corresponding to the schema. This help makes users understand more easily the schema definition. Then the *correspondence engine* takes pairs of schemas and generates correspondences between them, using an attribute classifier. To do so, for each attribute, *Clio* extracts features from small, random database samples. Then a Nave Bayes-based classifier finds similar attributes and suggests mappings between them. At the end of this step, the user can intervene too : he may add, modify or delete any relationship found by the *correspondence engine*. Finally the *mapping engine* gathers information from last engines to support the evolution and maintenance or to reuse old mappings. The final result of a mapping is a set of queries that take data from the sources and produce data conforming to the target schema. Depending on the source type, the queries are formulated in SQL, XQuery, or XSLT.

Clio has an user-friendly interface and is reliable to help the user during data integration

process. Besides it aims at providing many information so that the user might understand the reasoning done by the matching system and eventually correct some mistakes. This is a really interesting feature that should be included in most software, because the feedback could help to detect bugs or wrong reasoning. Another good point is the evolution and maintenance capabilities taken into account, even if the algorithms allowing these techniques are not detailed.

The machine learning algorithm used by Clio for matching is a good idea, but not sufficient to provide a good matching quality. It needs to be completed by other algorithms or a dictionary. Indeed with specific field domain, Clio could produce wrong mappings. More, the user has a too much important role and is advised to intervene at every step of the process. And there is nothing saying if the evolution and maintenance process is efficient at large scale. No experiments are shown to know the performances of Clio.

Chapter 3

ADOPTING A BTREE APPROACH TO SCHEMA MATCHING

As seen in the last chapter, most matchers currently give average results : their matching quality can be good, but they often offer slow performances and do not achieve to match correctly some specific cases. Our idea is to improve the performances by using the B-tree index structure, and from there to take advantage of the search speed in such structure to enable to check the ancestors of the elements before matching them. To demonstrate its performances, we designed a matching tool called Btree Match whose architecture is explained component by component. Furthermore we also focus on possibilities to reuse the B-tree structure in a peer-to-peer architecture.

Note that in this chapter, B-tree represents the structure that stores schemas elements, whereas Btree Match is the name of the matching tool.

This chapter is divided into five parts : the first one explains the B-tree, and how it works; next we give an overview of our application; the third part focuses on the details, for instance some algorithms or the architecture of the system; then a short example illustrates a step to step matching with Btree Match; finally we give some of the perspectives, especially for a large scale context.

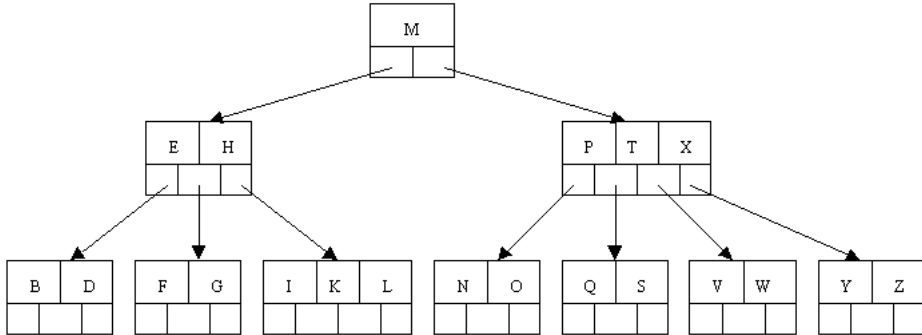


Figure 3.1: An example of B-tree of *order* 5.

3.1 Using B-trees

In our approach we use the B-tree as the main structure to find the mappings. The advantage of searching mappings in B-tree instead of in the schemas is that B-tree have indexes, which accelerates significantly this process. As described in [1], B-trees have many features. A B-tree is composed of nodes, each of them having a list of indexes. A B-tree of order M means that each node can have up to M children nodes and contains a maximum of $M-1$ indexes. Another feature concerns balancing, which means all the leaves are at the same level - thus enabling fast insertion and fast retrieval since a search algorithm in a B-tree of n nodes visits only $1+\log_M n$ nodes to retrieve an index. An example of B-tree from [11] is given in **Fig. 3.1** where indexes are letters sorted by alphabetical order.

We preferred using the B-tree rather than the B+tree since some searches are faster in a B-tree (for all indexes not in the leaves). Besides out B-tree tends to grow much so we don't need redundant indexes that appear in B+tree. Finally the main drawback of B-tree, namely the cost of deletion in a non-leaf node, cannot apply since we only add indexes. For those reasons, we choose to use B-trees in our matching algorithm.

3.2 Overview of Btree Match

Contrary to many others matchers, we have no pre-processing step in which we gather information about elements from the input schemas. The main idea is to build a B-tree from one of the input schemas. Once this is done, we parse each other input schema and integrate it into the B-tree. Finally we obtain the *mediated B-tree*, which represents the B-tree after having integrated all the input schemas. The mediated B-tree is then converted into an XML schema, simply by traversing it with a preorder traversal. Here we resume our approach to match k input schemas, where the *right arrow* represents the building of the second element from the first one, and the $+$ stands for the integration operation :

$$\begin{aligned}
 s_1 &\rightarrow \text{btree} \\
 \text{btree} &= \text{btree} + s_2 \\
 \text{btree} &= \text{btree} + s_3 \\
 &\dots \\
 \text{btree} &= \text{btree} + s_k \quad \text{btree} \rightarrow \text{mediated_schema}
 \end{aligned}$$

A B-tree needs unique indexes that will be used to sort it and facilitate the search. We choose to use for index the preorder path. This path is composed of the preorder values of each traversed node from the root node. The first element of the path is the schema number so that the path stays unique. **Fig. 3.2(a)** and **Fig. 3.2(b)** are examples of XML schemas from OASIS [25], and the paths that will be used as indexes in the B-tree are shown in **Fig. 3.3**. Some information are associated with each index : the name of the element, the path of the names from the root, which is used to find mappings, the type of the element, and a table with other attributes. There are some advantages to use the preorder path as indexes in the B-tree : first we are sure these are unique values. And when the B-tree is traversed using preorder traversal, each element appears obviously in the same order than when traversing the schema, so it is easier then to build the mediated schema from the B-tree.

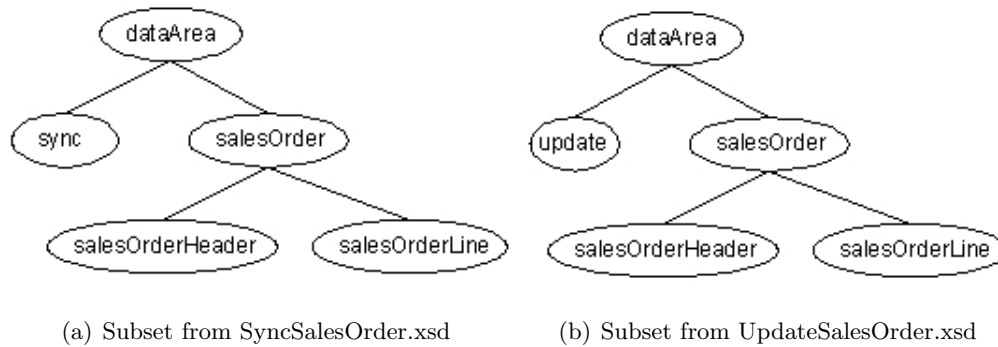


Figure 3.2: Examples of XML Schemas.

<i>Nodes</i>	<i>Paths of SyncSalesOrder (Fig. 1)</i>	<i>Paths of UpdateSalesOrder (Fig. 2)</i>
dataArea	1 / 1	2 / 1
sync	1 / 1 / 2	
update		2 / 1 / 2
salesOrder	1 / 1 / 3	2 / 1 / 3
salesOrderHeader	1 / 1 / 3 / 4	2 / 1 / 3 / 4
salesOrderLine	1 / 1 / 3 / 5	2 / 1 / 3 / 5

Figure 3.3: The paths of each node used as indexes in the B-tree.

3.3 Detailed discussion of the process

3.3.1 Describing the general method

First we choose the largest file among the input schemas, and it will become the basis of the B-tree. This enables to fill quickly the B-tree with the largest schema. After this first step, we need to parse the other schemas, one at a time, by using preorder traversal. For each node with a name, we search inside the B-tree if a mapping can be found. We compare the names of the nodes, and if they are equal, we check their ancestors in order to ensure that we do not map 2 elements which seem equivalent but does not refer to the same thing according to the context. For instance, let consider the 2 elements *date*, child of *author* with *date*, child of *book*. It is quite obvious that both elements does not represent the same thing when we consider their context (their parents in this case). This process can be fast thanks to the B-tree features. If no mapping can be found, the element is simply added in the B-tree. The algorithm illustrating this process is shown above in **Algorithm 1**.

In **Algorithm 2**, we describe how to match elements. This step is currently not enough accurate but will be improved in the future. We traverse the B-tree and compare each node with *currentElement*, the node to be matched. If there is a possible match with a node *n*, we compare the ancestors of node *n* in the *compareParent* method to check if *currentElement* and *n* are in the same context, meaning their ancestors have at least a relationship. The number of ancestors to be checked, called *NB_ANCESTORS_TO_GO*, is explained in **Chapter 4**. To sum up, our algorithm tries to find a relationship between *currentElement* ancestors and *n* ancestors until the *NB_ANCESTORS_TO_GOth* ancestor is reached. For the moment this relationship is only based on equality, but we intend to improve it too.

Algorithm 1 The parser algorithm which builds the B-tree.

for each parsed element

{

// updating preorders path

path = path + preorder + "/";

// analysing attributes of the element to find its name

for(each_attribute)

{

if(attribute_name == "name") keep the value to find mappings

else keep attribute in a list

}

extract name of the parent from the names path (pathName)

// updating the names path by adding the current parsed element name

pathName = pathName + name + "/";

if(building_b-tree) add the current element in the B-tree

else // matching process

{

call to searchMappings to search for mappings in the B-tree

if(mapping_found) store the mapping

else add the current element in the B-tree

}

// updating preorder value for next element

preorder++;

}

Algorithm 2 The *searchMappings* function that finds mappings between the current element and elements from the B-tree.

```

searchMappings(String currentElement, String parentName)
{
    for(each index ind)
    {
        if(ind.name == currentElement)
        {
            // we check the parents
            if(ind.compareParents(parentName) == true) return ind
        }
        // recursive calls to the children to search in the whole Btree
        if(left_child) left_child.searchMappings(currentElement, parentName)
        if(right_child) right_child.searchMappings(currentElement, parentName)
    }
}

compareParents(String parentName)
{
    for(int i=1;i<=NB_ANCESTORS_TO_GO;i++)
    {
        String parName=getAncestorName(i); // get the i(th) parent
        if(parName.equals(parentName)) return true; // compare wth the parent
    }
    return false;
}

```

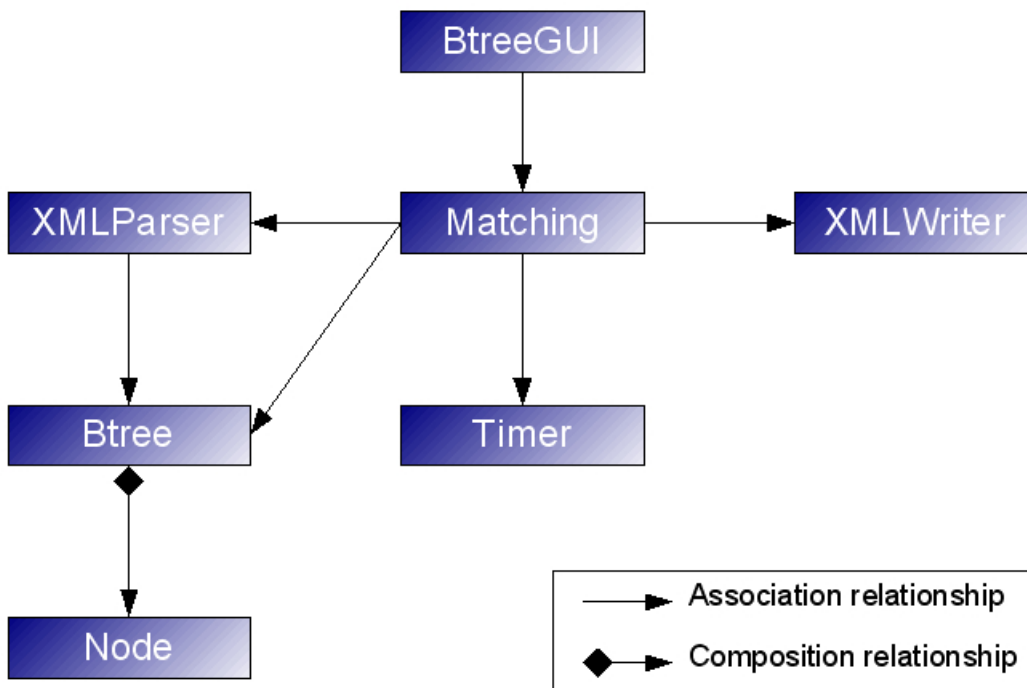


Figure 3.4: Architecture of Btree Match.

3.3.2 Btree Match

In this subsection we describe the matching tool. This application is composed of different modules, each of them having a well defined role. **Fig. 3.4** shows the relationships between these components.

Next we describe with more details each of this components :

- **BtreeGUI** is the starting point of the application. It is also the graphical interface with which the user interacts. More details about it are given further.
- **Matching** is the main module of our tool. It is in charge of initializing other components, like the Timer or XMLParser. It also chooses the biggest file among those to be matched and keeps the results of the matching, namely the mediated B-tree, the time elapsed and the mappings. If a new matching is launched, the current one is erased, although the generated mediated schema and the mappings might be saved in files.

- **Timer** counts the elapsed time of the matching, including the creation of the mediated XML schema.
- **XMLParser**, as its name indicates, is used to parse an XML schema and fill in the B-tree. Two distinct cases appear : if this is the first schema - so the biggest one -, each element is just put into the B-tree. Else with all other files selected for the matching, the parser try, for each parsed element, to find a match with elements already in the B-tree. In the end of the process, we obtain the mediated B-tree which is then returned to the Matching component.
- **XMLWriter** is called after the matching process. It creates the new mediated XML schema from the mediated B-tree, and also creates a mapping file. The mediated B-tree is traversed using preorder, and two stacks are needed to store open elements that will be closed when their children have all been parsed. This is a very easy and fast step, since the B-tree is already sorted.
- **Btree**, which represents the B-tree structure. It is composed only of Nodes, but references only the root node. All operations, for instance insertion or search, starts at the root node and reach deeper nodes if needed.
- **Node** contains its parent Node and a list of indexes, each index representing one element from the XML schemas. Thus each index stores the preorders path, the names path, and the element name, preorder value and type. Each index has also a left child and right child Nodes, which respectively leads to the elements accessed before and after the current index.

We had a quick overview of the architecture, now we concentrate on the features of Btree Match. Given that it was mainly designed to demonstrate the capacity of the B-tree structure to manage efficiently schemas elements while matching, the application shown in **Fig. 3.5** does not offer many functionalities. However, as the next perspective is to use these results in a matching peer-to-peer architecture, we think that some components of Btree Match might be easily reused, like the B-tree or the XMLParser.

At the very top we access the menu bar (noted 1 on the figure) : it enables to open a file which is restricted to XSD format, view the content of an XML schema or execute a matching. The interface is split into two panels : the top one (noted 2) lists files chosen by the user. Note that these files are just displayed so that the user might then select them, but no processing is done when opening the file. To view or match, the user must select some files from this list. The results of these actions are shown in the bottom panel (noted 3).

This interface is minimal, we could for example have added some buttons to allow the suppression of files in the list, or more useful a syntax coloring for the bottom panel or the possibility to show the mediated B-tree with a tree format¹. The parameters of the application should also be tuned from this interface and other schema formats than XSD could be opened². However the main goal was to prove the performances of the B-tree structure to handle a matching, and further work would probably use another system or architecture, enabling only some components to be re-used and not the whole application.

¹This is not primordial need but it could be done by an external developer.

²This step implies however to write an appropriate parser for the considered format.

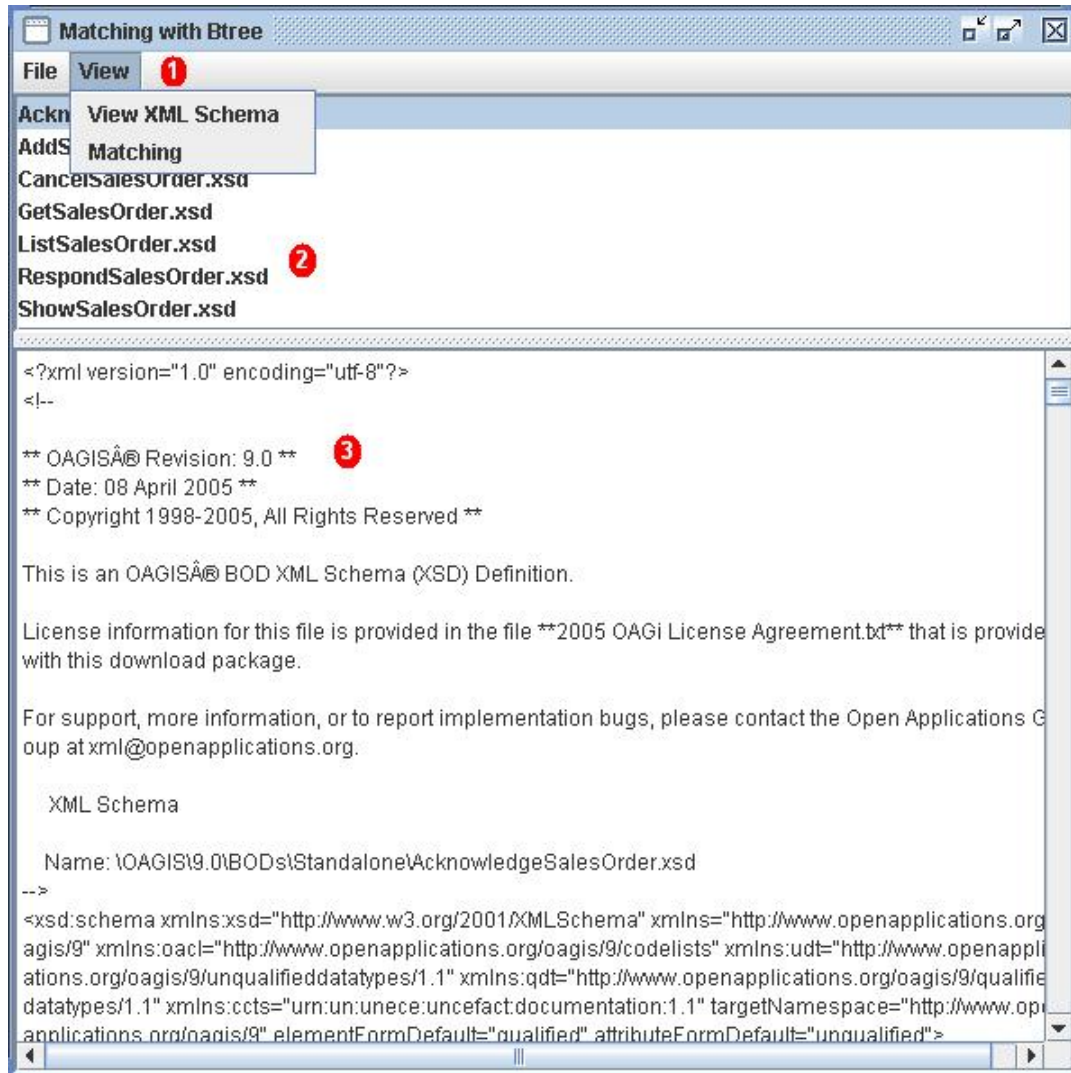


Figure 3.5: Screenshot of Btree Match.

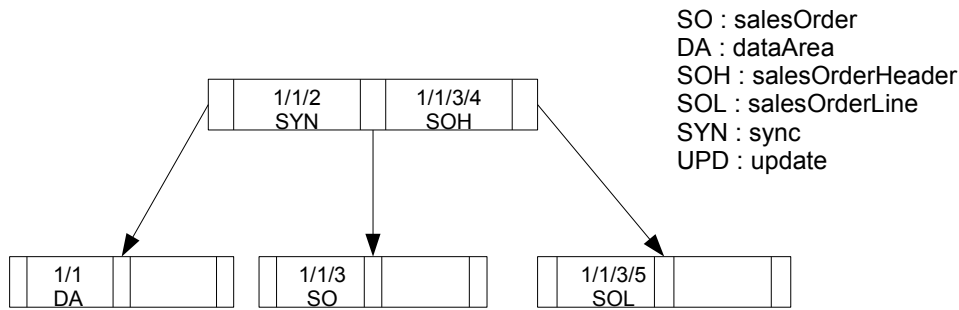


Figure 3.6: The B-tree after building it from *SyncSalesOrder.xsd*.

3.4 Example

We use the last schemas shown in **Fig. 3.2(a)** and **Fig. 3.2(b)** to illustrate how the B-tree is build and how the matching process occurs. Our algorithm chooses the bigger schema as the first one to initialize the B-tree. Let consider that *SyncSalesOrder.xsd* is the first schema. So we build the B-tree by adding an index for each of the schema node. This index is the path found by preorder traversal shown in **Fig. 3.3**, and other fields are associated like the name of the node, its type, etc. Some operations during insertion enable the B-tree to keep balanced, see [1] or [11] for more details. The result at the end of this process is shown in **Fig. 3.6**. At this stage, we have just parsed each element from the first bigger schema and added it into the B-tree.

Then we integrate *UpdateSalesOrder.xsd* : the first parsed element is *dataArea*. The algorithm finds in the B-tree a similar element in the left leaf, so it tries to check their parents. As those nodes are the root nodes, they have no parent but in our modelisation, the schema itself is considered as their parent thus the algorithm deduces a mapping between the two *dataArea* nodes. We add a reference of the second element in the *dataArea* node already present in the B-tree³. The second element is *update*, and we cannot find any matching with any elements in the B-tree, so we simply add it to the B-tree. As its preorder path starts

³This reference does not appear in the figures.

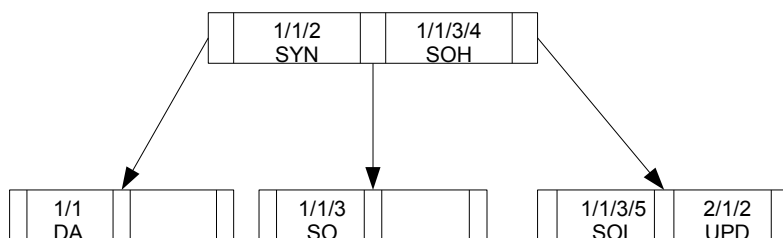


Figure 3.7: The B-tree after integrating *UpdateSalesOrder.xsd*.

with the schema number, this element is inserted after all elements of the first schema. Then the algorithm goes on with *salesOrder* and a match could be found. By checking their parents, both *salesOrder* parents have a relationship since *dataArea* nodes were matched earlier. We can also match the two *salesOrder* elements, and a reference is added in the one already present in the B-tree. We will not describe others elements because the process is the same but none of them is added in the B-tree since we can find a matching for each of them. They are just stored, and we finally obtain **Fig. 3.7**.

Concerning the mappings found, there are several possibilities to store them : in a text file, like in Charlie, in a repository⁴, like in COMA++, or directly in memory⁵. We decided to store them both in the B-tree, where they could then be accessible and useful, and in a text file so that a user can still retrieve them later if needed.

⁴More precisely, a database.

⁵In our case, inside the B-tree.

3.5 Support for dynamic processing

In a static environment, the schemas and discovered mappings does not become obsolete. On the contrary, in a dynamic context, the peers hosting the schemas may join or leave the network and the schemas might be modified by the users, implying the data sources to be updated to reflect these changes.

3.5.1 Improving matching quality

The first perspective concerns the matching quality. The algorithm takes into account the context of the ancestors but could be enhanced in many ways :

- **Dictionary or thesaurus use** obviously provide new possibilities : the synonyms and other semantic relationships could be found between elements, thus implying many mappings not found before to be correctly . This process should be completed by a cleaning process : the delimiters⁶ are used to decompose the element name into a table of names. With the dictionary, we then complete this table with a list of synonyms. Each element would keep its table, which then would be used to find mappings with other elements.
- **ALP⁷** could be used to describe a schema with keywords so that the algorithm does not parse any schema which is completely out of the requested domain.
- **Machine Learning techniques** might generate some models from which we could classify the mappings.
- **Cardinality** is currently restricted to match one element with another. Yet many cases show that one element match several elements, so our algorithm could be enhanced to reflect this reality.

⁶A delimiter is a character used to separate words, for example *author_name*, *author name* or *author-name*

⁷Automatic Language Processing

The other possibility to improve matching quality would be to integrate B-tree structure inside another matcher, for instance COMA++. Thus we would benefit of the speed of the B-tree and the good matching quality of COMA++. However this solution implies the source code of the matchers to be open-source, which is not currently the case.

3.5.2 *MetaIndex*

Most of the matchers currently keep the mappings found in a text file or in a repository as in COMA++, store them in memory or they just print them on the screen. Yet those mappings need to be used later since they should enable queries to find quickly the interesting results on every schema. Building the mediated schema is one thing, but storing the mappings anywhere can slow down the process of using them.

Our idea is to keep the mappings in the B-tree structure, in memory, where they could then be exploited easily. There would have no need anymore for a mediated schema, which anyway is impossible to build in a large scale environment. Each element could reference the elements it is mapped with, so they can be accessed quickly when needed. We called this *MetaIndex* since an index could store information about other indexes. It also enables to maintain coherence among the mappings since when a change⁸ occurs, we just have to alert the peers owners of the mapped elements so that they might reflect those changes in their own B-tree.

Obviously in our Btree Match, the B-tree structure keeps the node of many schema. However if we consider a peer-to-peer context, each peer will store only the elements of his own schema and eventually the mappings, so the B-trees of each peer will be smaller than the mediated B-tree in our Btree Match application. Thus the search for mappings will be very fast. Some work in [19] have already proposed to use independent B-tree structures maintained on each peer of a peer-to-peer network to facilitate the query rewriting. Therefore it should be possible to use a similar architecture including the *MetaIndex* to perform

⁸Namely a deletion or modification.

correct maintenance.

3.5.3 Refining

The search concerning the ancestors and the parameter *NB_ANCESTORS_TO_GO* could be more flexible. Indeed it could be set high, but can breakout after a number of levels of checking when it is clear that we do or do not have a match. This may results in better performances especially in large schemas. Besides, combined with dictionaries, the semantic distance⁹ between two elements could be calculated, and the breakout could then occur under a certain threshold.

The maintenance process involves the deletion of the mappings of a peer which left the network. Its mappings are effectively inconsistent. But what happens if the peer left for a few seconds, because of a network problem ? All its mappings need to be discovered again, and this would result in a loss of time. It could be a good idea not to delete the mappings as soon as a peer leaves the network, and just mark them as inconsistent. The inconsistent mappings might be kept for a certain time, and definitely deleted after this period. But if the peers comes back before the end of the period, it would not need to search for mappings again since other peers may communicate the last mappings found. This could be useful especially when the matching quality was really good.

On a peer-to-peer architecture, it is often common to have many computers unused, meaning that most resources are available for processing. The idea is to benefit from this unused resources to improve the matching quality. Future work may involve a search for mappings decomposed in levels : when a peer joins the network, a basic search for mappings is performed to discover them quickly. This first-level search should ensure that the discovered mappings are almost correct. Later on, a second search could be done again to improve the quality by using dictionaries or more sophisticated matching algorithms. Finally a third-level search could use, like in COMA++, a combination of several matching

⁹The semantic distance is a value showing the degree of similarity between two elements (0 stands for total similarity while ∞ means no similarity at all).

algorithms or some machine learning techniques to provide the best results. In this case, the process would be incremental, meaning that a peer which joined the network is quickly ready to answer some queries and will improve its matching quality with the time. We could even think further : if a peer needs to refine its mappings but is heavily used, it could let others computers search for mappings by sending them some information. There should have also the possibilities to mark the peers that have not a good matching quality according to the level of refining they have reached so far, and to consider them as less reliable. Thus we could imagine that the minimum semantic distance to be reached to match two elements could be higher for marked peers. On the contrary, it could be lower for peers that have reached a good matching quality, meaning they have already refined their mappings.

3.5.4 Conclusions

In this chapter we offer some improvements and future work for Btree Match. Some of them involves minor changes while others provide large perspectives that could lead to new projects. Obviously those ideas still need to be deeply thought, because many problems, like network and storage overloading or data privacy, can occur.

Chapter 4

RESULTS AND EXPERIMENTS

In this chapter, we explain we provide results and comparisons with the most used matching tool, COMA++, and another matcher which is specifically designed to improve performances, Charlie.

4.1 Parameters in Btree Match

There are 2 parameters in our application : the first one is the *ORDER*, which is used to calculate the number of children and indexes per node in the B-tree. In **Fig. 4.1**, the *ORDER* parameter value is modified to improve matching time. We can notice that the ideal value for the *ORDER* parameter stands between 20 and 30. Even with 40 input schemas, the experiments show that changing the *ORDER* parameter value either to 20, 50 or 100 has no significant impact on the performance.

The second parameter, *NB_ANCESTORS_TO_GO*, corresponds to the number of ancestors that are backwarded in order to find a more accurate mapping. For example, *NB_ANCESTORS_TO_GO* valued to 1 means we want to check only the parent node, *NB_ANCESTORS_TO_GO* valued to 2 means we want to compare with the parent and

	<i>5 Schemas matched 11 098 indexes, 1 565 mappings</i>	<i>10 Schemas matched 22 104 indexes, 4 149 mappings</i>
Order = 5	6 657 ms	20 918 ms
Order = 10	5 484 ms	18 094 ms
Order = 15	5 374 ms	17 782 ms
Order = 20	5 297 ms	17 656 ms
Order = 25	5 313 ms	17 547 ms
Order = 30	5 313 ms	17 641 ms

Figure 4.1: Varying *ORDER* parameter has no significant impact.

	<i>5 Schemas matched 11 098 indexes</i>	<i>10 Schemas matched 22 104 indexes</i>
NbAncestorsToGo = 1	7 142 ms / 2 086 mappings	25 220 ms / 4 670 mappings
NbAncestorsToGo = 2	7 203 ms / 2 107 mappings	25 846 ms / 4 734 mappings
NbAncestorsToGo = 3	7 219 ms / 2 107 mappings	26 032 ms / 4 734 mappings

Figure 4.2: *NB_ANCESTORS_TO_GO* parameter can provide better quality matching.

grandparent nodes, and so on. This parameter enables to find a kind of context, and to avoid the case where *author* \rightarrow *date* is matched to *book* \rightarrow *date*. Indeed the two *date* elements are not representing the same : one is probably the *date of birth* of the *author* whereas the other may stand for the *publishing date* of the *book*. Obviously the processing time increases when *NB_ANCESTORS_TO_GO* increases, but it is still negligible even with 10 input schemas and more. On the other hand, it can provide a better matching quality, as shown in **Fig. 4.2**.

To sum up, Btree Match has been designed to be flexible and allow the user to tune some parameters. One has no real impact on the performances but it enables to save some storage space in memory. Indeed, the more indexes a node contains, the less nodes in the B-tree. For instance, a schema composed of 10 elements would fit in only a root node in a B-tree of *ORDER* 20 while it would need 4 nodes in a B-tree of *ORDER* 3. The second parameter has directly an influence on the performances since it handles the number of ancestors that are searched inside the B-tree. As shown in the tests, a good compromise to keep correct performances and to enhance the matching quality needs only a low value, no more than 3.

4.2 Comparisons

In order to evaluate the performances of our Btree Match tool, several experiments enables to compare it to other matchers. All experiments were run using a 3GHz Pentium IV machine with 1GB memory on a Windows XP platform with Java virtual machine (JVM) version 1.5. The XML schemas are standards and have been provided by the Open Application Group [25]. Those schemas are quite big, about 250Kb each. The average depth in those schemas

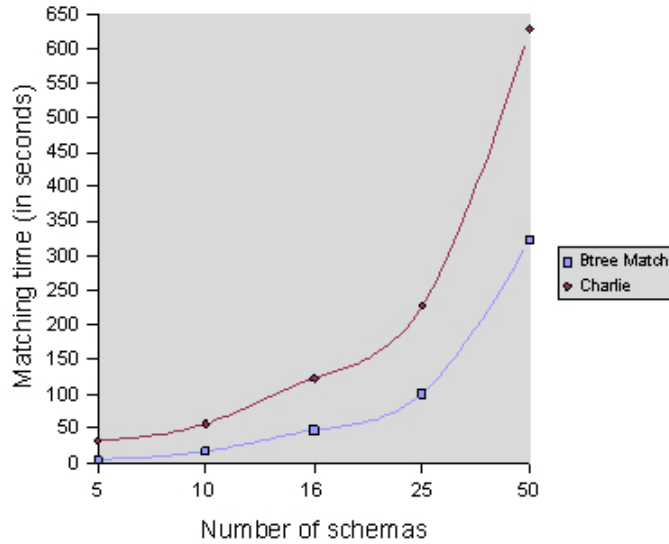


Figure 4.3: Matching comparison with Charlie on the number of schemas.

is between 8 and 10, and they have about 2000 elements. No bigger related-domain schemas could be found, but those used are sufficient to have reliable experiments. To measure the quality of the matching produced by the matchers, we examine the discovered mappings and check the structure of the mediated schemas.

4.2.1 With Charlie

On the next figures we compare the results with Charlie, which also aims at improving the matching time process. **Fig. 4.3** shows the time needed to match a certain number of schemas. Note that all XML schemas used in **Fig. 4.3** are about 250 KB each, the depth of those documents are between 8 and 21 and the number of elements of each schemas between 1900 and 6300. Our B-tree Match algorithm is better for the same quality of matching, if not better. In the next figure **Fig. 4.4**, we demonstrate that either with small or big XML schemas our algorithm is still better and offers at least the same matching quality. However we have noticed that our algorithm was not always efficient with small size schemas, maybe because we use the SAX parser which tends to be less efficient than DOM parsers on small documents.

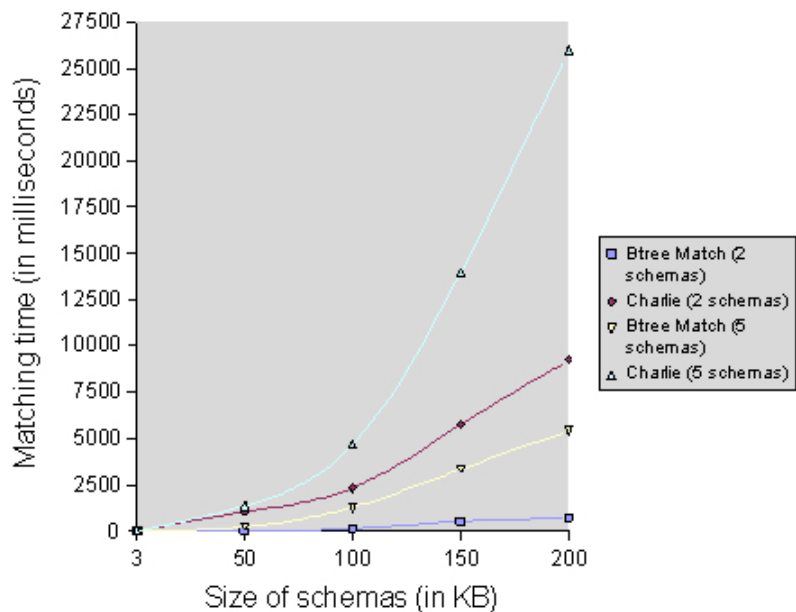


Figure 4.4: Matching comparison with Charlie on the size of schemas.

Charlie does not always provide a complete mediated schema and has not as good performances as the ones of Btree Match to match large schemas. Besides it does not offer any possibilities to use efficiently the discovered mappings, especially for a large scale context,

4.2.2 With COMA++

Finally we compare our work with COMA++, even if this last has not been designed for matching time performance but rather for matching quality. As COMA++ offers many strategies, we choose to use the easiest one and one existing matcher from the library. Under these conditions, it is still difficult to compare efficiently. In **Fig. 4.5**, we show that COMA++ finds more mappings, although it gives all mappings possibilities, but it also takes more time to match than with our method. Note that the time given by COMA++ is only the matching time, and includes neither the time needed to add the schema into

<i>Matching tool</i>	<i>Matching time (in milliseconds)</i>	<i>Number of mappings found</i>
Btree Match	1282	526
COMA++	26000*	2537

* only the matching time, and not the time to find information about schemas

Figure 4.5: Matching comparison with Coma++.

the repository nor the one to load the schema from the repository into a directed graph¹. Obviously COMA++ makes more processing than Btree Match, Thus the consequence is that using B-trees inside COMA++ could be a real improvement, benefiting from the accuracy of COMA++ and the speed of B-tree.

It is clear that our matching tool does not offer a matching quality as good as the one in COMA++. However our work needs to be continued and this comparison shows two perspectives. In one hand we could try to improve the matching quality of Btree Match by using dictionaries and thesaurus. On the other hand, we could enhance COMA++ by replacing its graph structure by a B-tree structure. This last idea involves to obtain the COMA++ source code, which is not currently available.

¹Adding those times implies several minutes more to execute those preprocessing steps.

Chapter 5

CONCLUSIONS

Schema matching is still an important problem today : the heterogeneous data sources and the growing quantity of information require fast algorithms to find similarities between elements of different schemas. Among the different languages used to describe the data, XML quickly spread due to its flexibility, extensibility and both human and machine readable format. So the problem is often reduced to XML schema matching. Much work has been done to solve its two subproblems, the matching quality and performances, but the results are not sufficient enough : the **Chapter 2** showed that the process is still semi-automatic, thus implying a human intervention. Furthermore the execution time is too slow and does not offer any perspectives for a large scale environment.

In this thesis, we presented our approach to improve the time elapsed on schema matching. We designed a B-tree schema matcher. The algorithm uses a B-tree structure which includes an index mechanism. With the fast access to the nodes of this B-tree we could afford to focus on the context of a node by checking its ancestors before validating a mapping. We then developed a tool, Btree Match, which is implemented in Java. It offers good results, especially with a large number of big-sized schemas. Then we ran a series of experiments. When compared with Charlie, the matching performances are far better. However it still suffers from a good matching quality, as the COMA++ comparison showed. As the main aim of the work was to improve matching performances, we can conclude that it is a success. Thus B-tree appears suited to large and numerous schemas, and should also provide good results in a large scale context like peer-to-peer networks.

Although our matching tool offers good results, this is not sufficient so that it is fully exploited. One could improve on our approach in different ways. The first possibility is to

enhance further the matching process, either by including an ontology or a dictionary like Wordnet or by integrating B-tree match into COMA++ to find mappings faster. B-tree nodes could also directly integrate semantic information to provide better quality. A second possibility could examine the large scale issue : Btree Match, as illustrated in **Chapter 4**, is suited to peer-to-peer schema matching where we can forget the mediated schemas and just keep an efficient structure like B-trees to find the mappings on-the-fly. Finally Btree Match may also be interesting to maintain dynamically the mappings in case of updates or deletions of some nodes.

BIBLIOGRAPHY

- [1] D. Comer : The Ubiquitous Btree, *Computing Surveys*, June 1979.
- [2] D. Aumuller et al : Schema and ontology matching with COMA++, *ACM SIGMOD*, June 2005.
- [3] H. Do, E. Rahm : COMA, a system for flexible combination of schema matching approaches, *Proceedings of the 28th VLDB Conference*, 2002.
- [4] J. Madhavan et al : Generic Schema Matching with Cupid, *Proceedings of the 27th VLDB Conference*, 2001.
- [5] P. Bernstein, E. Rahm : A survey of approaches to automatic schema matching, *VLDB Journal 10 : 334-350*, 2001.
- [6] Cong Yu and Lucian Popa : Semantic adaptation when schemas evolve, *Proceedings of the 31th VLDB Conference*, 2005.
- [7] Z. Bellahsene and M. Roantree : Querying Distributed Data in a Super-peer based Architecture, 2004.
- [8] A. Dohan et al : Learning to Map between Ontologies on the Semantic Web, *WWW 2002*, May 2002.
- [9] R. J. Miller et al : The Clio Project : managing heterogeneity, (<http://www.almaden.ibm.com/cs/clio/papers/SigmodRecord2001.pdf>), 2001.
- [10] L. Popa et al : Mapping XML and relational schemas with Clio, *ICDE 2002* , 2002.
- [11] Website of Computing and Information Science, Saint Vincent College. (<http://cis.stvincent.edu/swd/btree/btree.html>).
- [12] C. Batini et al : A comparative analysis of methodologies for database schema integration, *ACM*, December 1986.
- [13] J. Tranier et al : Where's Charlie: Family-Based Heuristics for Peer-to-Peer Schema Integration, *IDEAS 2004: 227-235*.
- [14] W. Su et al : Holistic Query Interface Matching using Parallel Schema Matching, *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

- [15] P. Bouquet, S. Zanobini : A Formal Theory of Schema Matching.
- [16] R. McCann et al : Mapping Maintenance for Data Integration Systems, *Proceedings of the 31th VLDB Conference*, 2005.
- [17] M. Sayyadian et al : Tuning Schema Matching Software using Synthetic Scenarios, *Proceedings of the 31th VLDB Conference*, 2005.
- [18] S. O. Yahia, S. Lorient : Interrogation flexible utilisant une arbre B+ flou, 2006.
- [19] A. Crainiceanu et al : Querying Peer-to-Peer Networks Using P-trees, *WebDB 2004*, June 2004.
- [20] Do Hong Dai, Schema Matching and Mapping-based Data Integration, *Ph.d. Thesis*, August 2005.
- [21] George A. Miller : Wordnet, a lexical database for English, *Communications of the ACM*, November 1995.
- [22] S. Bergamaschi et al : The MOMIS System, *Manual*, 2004.
- [23] D. Manakanatas and D. Plexousakis : A Tool for Semi-Automated semantic schema mapping : Design and Implementation, *DISWeb'06*, June 2006.
- [24] M. O'Connor : Level-based Indexing for Optimising XML Queries, *Master Thesis*, June 2005.
- [25] Organization for the Advancement of Structured Information Standards, (<http://www.openapplications.org>).
- [26] Wordnet, (<http://wordnet.princeton.edu>).
- [27] P. Valduriez : Some hints to improve writing of technical papers, May 1994.
- [28] Wikipedia, the free encyclopedia, (<http://www.wikipedia.org>).