Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis: TD-5/01

Extracting Typed Values from Semistructured Databases

Paolo Manghi

December 2001

Thesis supervisor:
Prof. Giorgio Ghelli

Abstract

To date, current investigations on *Semi-Structured Data* (SSD) have focused on query languages that operate directly on graph-based data by matching flexible path expressions against the graph-database topology. The problem with this approach is that it renounces in principle the benefits typically associated with typing information. In particular, storage and query optimisation techniques, representation of user-knowledge of the data, and validation of computations cannot be based upon static typing information. On the other hand, the various attempts to reintroduce types for SSD, while effectively returning some of the benefits of static typing, compromise the irregular nature of SSD databases, by allowing just for mild forms of irregularities.

Our investigation is motivated by the observation that, despite the inherent irregularity of the structure, many or indeed most SSD databases contain one or more subsets that present a high degree of regularity and could therefore be treated as typed values of a programming language. In this thesis we lay the formal foundations underlying a novel query methodology based on an *extraction system* that, given an SSD database and a type of a target language, results in: (i) a subset of the database that is semantically equivalent to a value of the given type; (ii) a measure that informs the user about the quality of his type with respect to the original database. The extracted subset can then be *converted* into a value of that type and injected into the language environment, where it can be computed over with all the benefits of static typing.

To my family,
and especially to Carla and Rosanna,
whose smiles I will never forget.

Mountains should be climbed with as little effort as possible and without desire. The reality of your own nature should determine the speed. If you become restless, speed up. If you become winded, slow down. You climb the mountain in an equilibrium between restlessness and exhaustion. Then, when you're no longer thinking ahead, each footstep isn't just a means to an end but a unique event in itself.

Robert M. Pirsig
"Zen and the Art of Motorcycle Maintenance".

Acknowledgments

Many people have directly or indirectly partaken in making this thesis possible.

To begin, Prof. Giorgio Ghelli, Prof. Richard Connor, Prof. Antonio Albano, and Fabio Simeoni deserve my special gratitude. Their support, advice, and ideas have been fundamental for the development of this work. Similarly, I wish to thank Prof. Fabio Crestani and Prof. Malcolm Atkinson, whose precious suggestions have contributed to its improvement.

For the uncountable discussions on computer science and life we had in the last four years, I also wish to thank my friends and colleagues Fabio, Vincenzo, Nadia, Valentina, Giuseppe, Gabriele, Anna, Dario, Carlo, Keith, David, and Steve.

I also owe a special thanks to my closest friends, who gave me evidence that spatio-temporal distances are just a state of mind.

Finally, I wish to express my gratitude to Federica and my numerous family for their patience and support. Their invaluable contribute has been to remind me daily that personal life and computer science should have nothing to do with each other.

Contents

1 Introduction **1**
1.1 Thesis goals . . . 3
1.2 Thesis outline . . . 4

2 Semistructured Data **5**
2.1 Database construction . . . 6
2.1.1 Schema-first approaches . . . 7
2.1.2 Data-first approaches . . . 8
2.2 Schemas . . . 11
2.2.1 Schema benefits . . . 11
2.2.2 Optimal schemas . . . 13
2.3 Irregular information sources . . . 14
2.3.1 Irregular problem domains . . . 14
2.3.2 Irregular data sources . . . 15
2.4 Semistructured Data . . . 16
2.5 Semistructured data manipulation . . . 19
2.5.1 Query languages . . . 19
2.5.2 Query languages implementation . . . 20
2.6 Semistructured data drawbacks . . . 21

3 Typing semistructured data **23**
3.1 Static and dynamic typing for SSD . . . 24
3.1.1 Static typing . . . 24
3.1.2 Dynamic typing . . . 33

4 Extraction Mechanism **35**
4.1 Intuitions . . . 35
4.2 Extraction mechanisms realisation . . . 38
4.2.1 Extractability . . . 38
4.2.2 Extraction algorithm correctness . . . 39
4.3 A new SSD query methodology . . . 41
4.3.1 Possible application scenarios . . . 41
4.3.2 Implementation notes . . . 42

4.3.3 Comparison with other typing techniques for SSD 43

5 A Semistructured Data Model 45
5.1 A formal definition of SSDBs 46
5.2 Inclusion of SSDB graphs 48
5.2.1 Inclusion by labeling and topology 48
5.2.2 Identity by bisimulation 48
5.2.3 SSDB d-inclusion 49

6 Target language 51
6.1 Type language 51
6.1.1 Recursive types 53
6.2 Type equivalence 54
6.2.1 Weak type equality 54
6.2.2 Strong Type Equality 56
6.3 Value language 60
6.4 Mapping from values of L onto SSDBs 62
6.5 Definition of typing 63
6.5.1 Observation about typing 64
6.6 Axiomatisation of typing 68
6.6.1 Completeness 70
6.6.2 Soundness 84

7 Extraction algorithm 87
7.1 Extractability for L 87
7.2 The code 87
7.2.1 Generation of marked edges 89
7.2.2 Assumption sets 92
7.2.3 Reading the algorithm 94
7.3 Termination 95
7.4 Relevance 96
7.5 Cost 101

8 Extraction Algorithm Correctness 105
8.1 Soundness 105
8.1.1 Soundness of inclusion 106
8.1.2 Soundness of typing 109
8.2 Completeness 122
8.2.1 Cases of Incompleteness 122
8.2.2 Case of completeness 127

9 SNAQue **129**
9.1 The prototype 129
9.2 From XML documents to SSDBs 132
9.2.1 Ordering 132
9.2.2 Attributes 132
9.2.3 Elements with mixed content 133
9.2.4 Empty Elements 134
9.3 From CORBA IDL to types of L 135
9.3.1 Sequences and Unordered Collections 137
9.3.2 Unions 137
9.4 CORBA object instantiation 138
9.5 Experience with the prototype 141

10 Conclusions and Future Issues **145**
10.1 Customised extraction mechanisms 147
10.2 Persistence and extraction mechanisms 147
10.3 Databases data-first design 148

Bibliography **149**

List of Figures

1.1 Equivalence of expressiveness between values of a typed language and SSDBs. 2

2.1 From the real world to a database 8
2.2 From electronic data to a database 10
2.3 OEM and XML SSDBs of the Fibonacci's group pages 18

3.1 Type description for the value representing the Fibonacci's SSDB . . 26
3.2 UnQL database 31
3.3 Graph schemas 32

4.1 SSDB as expressive as the language value d. 36
4.2 Extraction of regular subsets 37
4.3 Inclusion relation 38
4.4 Mapping from typable values to SSDBs 39
4.5 Extraction algorithm outline 40
4.6 Typing relation 40

5.1 Graphical representation of a SSDB according to our data model . . . 46
5.2 lt-included and bisimilar graphs 49
5.3 Other forms of inclusion. 50

6.1 Weak type equivalence rules 55
6.2 Examples of values 60
6.3 Value d 64
6.4 SSDBs of Adam and Eve's family 66
6.5 Ambiguous typing 67
6.6 Ambiguous typing due to empty collections 68
6.7 Algorithm `Proof` 71

7.1 Extraction algorithm for SSDBs 88
7.2 Example of extraction 89
7.3 Example of wrong extraction due to loose termination test. 92
7.4 Example of extraction with a low-relevance type 97
7.5 Worst case SSDB and type 103

8.1 Example of incompleteness due to union types 123
8.2 Example of incompleteness due to edges with the same label 125
8.3 Example of incompleteness due to over-restrictive termination test . . 126
8.4 Extraction algorithm for tree-structured SSDBs 128

9.1 SNAQue: extraction of CORBA objects from XML SSDBs. 131
9.2 Example of mapping for attribute. 133
9.3 Example of mixed elements 134
9.4 Example of graph representation for mixed elements. 134
9.5 Example of mapping for attribute of text elements. 135
9.6 Example of graph representation for empty elements 135
9.7 An example of our mapping from XML onto graphs. 136
9.8 Example of main definition. 137
9.9 Extracted value 139
9.10 XML source produced by the parser. 142
9.11 Extracted value d 143
9.12 Client query in Java. 143

Chapter 1

Introduction

Database Management Systems (*DBMSs*) have proven to be extremely effective tools for the automatic management of *information sources*. Their efficacy and effectiveness relies on a number of fundamental assumptions, whose adequacy ultimately depends on specific features of the information source involved. However, when such features do not show, designers and users spend their time and energy building and using inevitably inadequate database systems. As an example of this, consider information sources with frequently changing structure. Schemas of DBMSs handling such sources should mirror these structural changes, and users should recast old data and applications to fit the new schemas.

Over the past few years, there has been an increasing interest in information sources that are too *structurally irregular* to be effectively handled by traditional DBMSs. This inadequacy has called for novel data models and query languages, and has led to the realisation of *Semi-Structured Database Management Systems* (*SSDBMSs*).

In SSDBMSs, *semistructured databases* (*SSDBs*) represent information sources as rooted, labelled graphs. The main characteristic of these *semistructured data models* is the integration of the traditionally separate concepts of schema and data into a single, flexible data structure. This way users can insert arbitrarily structured data into the database, with no concerns about a separate, pre-defined schema. In addition, data can be successively retrieved by referring to the meta-information provided by the labels.

We shall collectively refer to SSDBs as *semi-structured* data (*SSD*) or *self-describing* data, for meta-information is intermixed with data and does not appear as a separate entity.

The query methodology underlying most SSDBMSs relies on SQL-like query languages. Informally, a query specifies a set of *path expressions*, which are a set of labelled paths to be matched against the graph topology of an SSDB. It then returns the subset of the SSDB which reflects the structure identified by the path expressions.

The major drawback of these *navigational* approaches is that SSDBMSs intrinsically disown the general benefits typically associated with static typing in DBMSs. Indeed, due to the absence of a pre-defined schema, data access and query optimisation techniques cannot be supported, user knowledge of the data is harder to acquire, and query correctness cannot be guaranteed. In addition, programmers are forced to write applications in the low-level algebra of labelled graphs.

Attempts to recover some of the advantages of static typing by reintroducing the concept of schema for SSD have limited applicability. The efficacy of these techniques degrades in the presence of information sources with a considerable amount of irregularities, where the only reasonable solution seems to be the navigational approach.

Our investigation is motivated by the observation that, due to the flexibility of labelled graphs, SSDBs may also represent regular information sources, such as those typically represented by the values of a typed language. Intuitively, as illustrated in Figure 1.1, *regular* SSDBs could be thus conveniently converted into the equivalent typed values and computed over under the governance of the language's static typing regime.

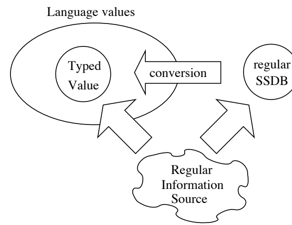

Figure 1.1: Equivalence of expressiveness between values of a typed language and SSDBs.

As SSDBs are usually adopted to represent irregular information sources, this observation is apparently of no practical value. Despite the irregularity of the structure, however, many or indeed most SSDBs "contain" one or more regular SSDBs. We are specifically interested in *identifying* the regular subsets of an SSDB that are equivalent to language values of a given type. When these subsets can be identified, we can *generate* the corresponding values and operate over them in a typed language.

Our approach is independent from the amount of irregularity of an SSDB, and aims at recovering all benefits of static typing whenever this may be convenient. Due to their complementary focus, in particular, we believe that our approach can be combined with the navigational ones in a system that offers complete support

for the management of SSDBs. Consider, for example, the management of irregular information sources with a large, structurally regular core. In our system, such sources would be represented by SSDBs to be indifferently computed over by navigational queries, for operations potentially involving data irregularities, and by typed applications, for operations regarding the regular core of the SSDB.

In its complete version, the system could be used with the further purpose of enabling typed applications to safely *update* SSDBs. The problem, not investigated in this work, is that of modifying the original SSDB according to the modifications carried out to the extracted values.

1.1 Thesis goals

This thesis is about the realisation of *extraction mechanisms* for typed programming languages.

First, we provide a specification for the extraction process, according to which any language can be associated with a notion of *extractability*. Extractability for a language captures the concept of value extractable from an SSDB according to a given type. Based on extractability a corresponding algorithm can thus be constructed and proved correct.

We then show the feasibility of extraction mechanisms for most typed programming language in use. To achieve this, we first define the set S of SSDBs, and then characterise and implement an extraction mechanism for a representative language L. The definition of L consists of a typing relation between a set D of values and a type language $\mathbf{T}$, where $\mathbf{T}$ comprises a set of standard types: atomic, record, collection, union, and recursive types.

We believe that the generality of L entails an informal proof of the feasibility of an extraction mechanism for all typed languages that support at least a subset of the types in $\mathbf{T}$. Furthermore, other types could find a suitable mapping in S and more specific extraction mechanism could be devised.

Specifically, we formally characterise extractability for L. A value $d \in D$ is *extractable* from an SSDB $s \in S$ according to a type $T \in \mathbf{T}$, if d is *of type* T and there exists an $s' \in S$ *included* in s such that s' is *equivalent* to d.

Based on extractability, we provide a corresponding algorithm `Extraction` for L. Given s and T, the algorithm returns a value d extractable from s according to T, if one exists, or else it fails. In case of success, `Extraction` returns also a measure of quality of the extraction process. Such measure, called *precision*,[1] quantifies the practical value of d in terms of the amount of information in s that is potentially relevant to T but not extracted in d. Precision may help users at defining a better input type, and thus extract a larger and more useful subset of data.

[1] Not to be confused with the notion of precision in information retrieval.

Moreover, we formally prove that `Extraction` is *sound* with respect to extractability of L. This ensures that if `Extraction`(s, T) returns d, then d is extractable from s according to T. We also show that the algorithm is not complete with respect to extractability. Accordingly, `Extraction`(s, T) may fail even if there exists a value extractable from s according to T. We discuss the consequences of this in terms of the usability of the algorithm. To conclude, we define a complete algorithm $\texttt{Extraction}_t$ for the extraction of values of L from *tree-structured* SSDBs.

Finally, we shall illustrate the applicability of extraction mechanisms by presenting the system *SNAQue*. SNAQue enables CORBA-compliant languages to compute over regular subsets of XML SSDBs.

1.2 Thesis outline

The work is organised as follows. The first two chapters give an overview of SSD research. Specifically, Chapter 2 focuses on the main motivations behind SSD introduction, and presents semistructured data models and query languages. Chapter 3 completes the SSD survey by discussing the techniques proposed in the literature to overcome the absence of a schema in SSDBs.

The general principles underlying extraction mechanisms, along with possible applications and implementations of the mechanisms, are described in Chapter 4. The subsequent chapters define the extraction mechanism for the language L. In particular, Chapter 5 provides a definition of the SSD domain we shall refer to in our study. Chapter 6 defines the representative typed language L, providing a type language, a type equivalence relation, a set of values, and a typing relation. The mapping from language values to SSDBs is also defined here.

Chapter 7 shows an extraction algorithm `Extraction` for L based on the notion of extractability for L. The notion of precision of extraction is defined and termination of the algorithm is proven. In Chapter 8, the proof of soundness for the algorithm `Extraction` is illustrated and the general problems behind completeness of `Extraction` are extensively discussed.

Chapter 9 shows a practical application of an extraction mechanism by reporting the results of a project called *Strathclyde Novel Architecture for Querying document Exchange format* (*SNAQue*). A prototype of SNAQue has been developed at the Department of Computer Science of the University of Strathclyde, Glasgow (UK).

Chapter 2

Semistructured Data

In the last few years, there has been an increasing interest in storing, handling and querying *semistructured data*. The literature does not provide a unique clear definition of semistructured data, which have been indifferently referred to as *data whose structure is not known in advance*, *data stored out of a database*, *XML files*, *data on the Web*, *data with irregular structure*, etc.

In this thesis, we refer to a definition of semistructured data which abstracts from concepts such as regularity or irregularity of the structure and only depends on the data models through which the data are represented. Moreover, we show the motivations behind the introduction of semistructured data models, which ground on the inappropriateness of *Database Management Systems* (DBMSs) to handle peculiar kind of information. We believe that separating the data models from the information sources to be handled can provide a strong reference for understanding the rational behind semistructured data research.

DBMSs offer services for the efficient handling of *information sources*. An information source is a collection of information characterised by a specific internal organisation, i.e. a *structure*, which can be exploited to identify specific portions of the current *instance* of the collection. In DBMSs, a *schema* is the representation of the structure of the generic instance of an information source, while a *database* represents a specific instance of an information source.

In this Chapter we shall explore the relation between the definition of a schema and quality and efficiency of the corresponding database, to find out that DBMSs are no longer convenient when the schema is not *optimal*. Indeed, we shall see that a useful schema describes *regular* data, thereby providing high-quality modelling of the information source involved, supporting high-performance operations over the corresponding database, and guaranteeing correctness of applications. Vice versa, a non optimal schema generally describes *irregular* data, thereby providing the same benefits with unreasonable human and system costs.

We shall define *irregular (regular) information sources* those information sources whose structure cannot (can) be described by an optimal schema in a DBMS, and

whose instances are hence represented by irregular (regular) data.

The inappropriateness of DBMSs to handle semistructured information sources, called out for novel technologies. Investigations led to the definition of SSDBMSs and *semistructured data models*, according to which any information source is represented as a rooted, directed, labelled graph carrying values on the leaves. Accordingly, in SSDBMSs a database, namely a *semistructured database* (*SSDB*), is a collection of electronic data representing both structure and instance of an information source. We shall define as *semistructured data* (*SSD*) any collection of electronic data forming an SSDB.

Note that, differently from traditional data models, which provide type languages to define expressive user-defined schemas, SSD models offer only a single predefined data structure to represent SSDBs. The choice of a graph-like structure has a twofold advantage: labelled graphs can model any sort of information source, and data can be inserted into the database at any time, with no restrictions on the structure of the data. At the same human and system costs, an SSDB may represent regular or irregular information sources.

On the other hand, the graph-structure of SSDBs inevitably impoverishes the level of interaction with the data for both users and system. Indeed, *semistructured query languages* support commands to run queries over the graph structure of a database, so as to return smaller, possibly more regular views of the original database. Hence, users are matter of factly operating only over graphs, as no other data structure is available. Most importantly, due to the lack of a schema, i.e. an explicit description of the database's content, the underlying system cannot support the typical benefits associated with static typing in DBMSs, even in presence of regular data.

2.1 Database construction

The main motivations behind the introduction of SSD research are to be found in the construction process of a traditional database, which leads from an information source to a correspondent database. We identify two approaches to traditional database construction, namely *schema first* and *data-first*, whose difference stays in the kind of information source involved.

- In schema-first approaches the information source is a reality to be electronically organised, namely a *problem domain*. The realisation of a database consists of a precise sequence of stages involving the *modelling* of the reality and the *implementation* of the resulting model, with particular attention to the modelling of the structure of the problem domain and its relative implementation through a database *schema*. The database, representing a particular instance of the problem domain, is populated only on a second stage.

- In data-first approaches the information sources are electronic data available before the database is constructed. Data engineering analysis must be performed, in order to realise the database schema to which the interesting subset of the external data source will conform, as well as the software required to automatically populate the database with that source.

2.1.1 Schema-first approaches

In schema-first approaches (see Figure 2.1) a database is realised according to a long and engaging work, which begins with the analysis and modelling of the structure of the information source of a reality, namely a *problem domain*, and ends with a *problem implementation* which satisfies at best the requirements of designers.

Usually, a generic instance of the problem domain can be conceived as a set of *entities* characterised by a specific *structure*. According to this structure, entities are associated with a set of *properties*, which are facts describing a feature of an entity, and can be classified in *categories*, each gathering entities featuring the same set of properties. Modelling, in database design, consists in identifying the set of categories of a generic instance of the problem domain by means of the *abstraction mechanisms* of a given *data model*. A *conceptual model* is the result of modelling a problem domain according to a given data model.

Consider the problem domain of a library, where the category of *books* contains entities corresponding to individual *books*, each characterised by the properties *title* and *authors*. An analyst, working with object-oriented data model abstraction mechanisms, would model this problem domain as a *class Books* of *objects* with *properties title* and *authors*. Similarly, using relational data model abstraction mechanisms, the analyst would have represented this problem domain as a *relation* with *attributes title* and *authors*.

A conceptual model becomes a database when implemented through a computer language embodying the abstraction mechanisms of a particular data model. The resulting system encompasses schema, data, and applications as described by the conceptual model and is called *problem implementation*.[1]

As mentioned above, the main issue of conceptual models is that of providing a non-ambiguous description of the structural organisation of the entities of a problem domain. On the same line, problem implementations focus on an accurate realisation of a *schema*, commonly intended as a computer description of the structure of the data that will be hosted in the database. The name schema-first is due to the fact that database population, i.e. the operation of creating the electronic data representing the current instance of a problem domain, takes place after the creation of the schema, i.e. the representation of the structure of the problem domain.

[1]Note that conceptual model and problem implementation may be based on different data models. Mappings from data models into others are available, allowing the realisation of problem implementations which are sound with respect to conceptual models based on different data models.

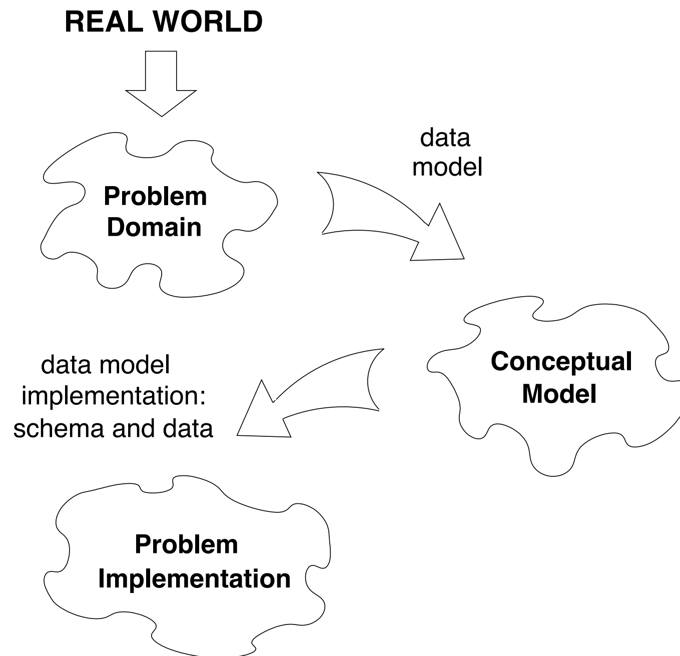

Figure 2.1: From the real world to a database

2.1.2 Data-first approaches

In data-first approaches, from which the name, the problem is that of developing a database to handle a subset of data stored in a set of given electronic information sources (see Figure 2.2).

These data sources can be classified as *databases* or *documents*. Databases contain data whose purpose is that of being handled with query languages, while documents contain data whose main purpose is that of being crated, modified, and eventually visualised by means of specific applications.

Unlike schema-first approaches, in data-first methodologies there is no need for a proper modelling phase as the information sources are generally characterised by either an explicit description of the structure, i.e. the schema of the database, or an implicit structure, i.e. a precise textual pattern which outlines the relevant

information in a document. Instead, an engineering analysis of the data sources involved is required, in order to define a schema mirroring the structure of the subset of data of interest.

Once the schema has been constructed, the corresponding database is ready to be populated with the relevant set of data. This operation involves the transformation of the relevant subset of the data sources into data instances conforming to the schema of the target database. Such transformation is performed by ad-hoc software, namely *wrappers* [7, 8, 69, 43, 18, 80], which retrieves the relevant subset of data from the data sources to insert it into the target database according to its new structure.

In general, the realisation of wrappers is a non-trivial and ad-hoc task. In particular, observe that wrappers moving data from a source database into a target one, rely on DBMSs primitives for both extracting and inserting data. Instead, wrappers that move data from documents into a target database, are in fact parsers which search for the data identified by a given textual pattern and then insert them into the target database through the appropriate primitives. Next, we shall see that there are various classes of documents, and point out which of these are more suitable for data-first approaches.

Data in documents

Nowadays, a large amount of the information processed when working on a computer is stored in *documents* rather than in databases. Consider for example the World Wide Web, which is the greatest repository of information on earth. Data on the Web is stored in documents, and the only way to access the information therein is typically through visualisation and reading. Needless to tell, if inserted into a database, this information could be queried over with great advantage for system users.

To achieve this, users apply data-first approaches: they analyse their documents, identify a textual pattern which corresponds to a DBMS data structure, realise a DBMS schema according to that pattern, and develop the corresponding wrappers. However, this process is not applicable to all kinds of documents, some of which may be illegible, hence not analysable, by humans. In particular, we can identify two main categories of documents:

- *Interpreted documents*: this class gathers documents that can only be interpreted by the applications that created them. Examples are *zip* files, *ps* files, pictures, and sound, which can be modified and/or visualised and/or executed only through specific applications, such as *Win-Zip*, *Ghostview*, *Photoshop* and so on. In summary, the storage format of interpreted documents focuses on how to represent fonts, styles, characters, colors, pixels, and so on, and is hardly readable by a human.

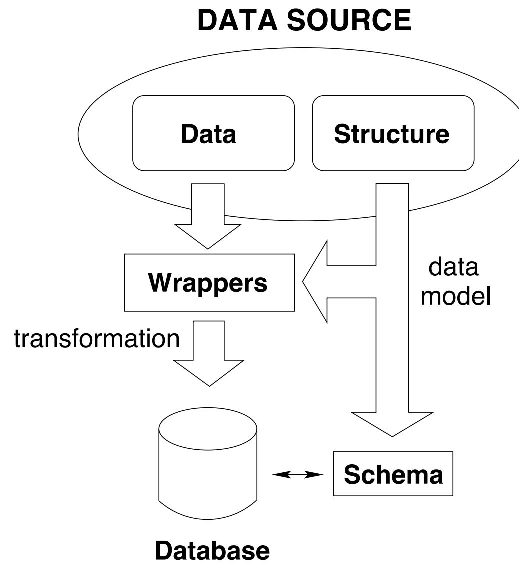

Figure 2.2: From electronic data to a database

- *Legible documents*: this class gathers documents whose main purpose is that of being modified and visualised for consultation by humans.[2] Well known instances of these documents are BibTex files, HTML files, XML files, digital libraries, on-line documentation, e-commerce database files and so on. Of course, legible documents may also be interpreted by specific applications, such as *BibTex, Latex*, and *Web browsers*, but their storage format outlines a human readable content [37, 1].

Data-first approaches are applicable to legible documents only, which, unlike interpreted documents, can be read and manipulated by humans.

Consider for example BibTex files, whose content could be queried with great advantage for documents writers. BibTex files data closely resemble relational data, as they consist of a set of entries that could be mapped onto a list of records. Hence,

[2]It is hard to trace a neat line between interpreted and legible documents. For instance, is a Word file interpreted or legible? The answer is up to the human ability of interpreting the underlying structure of a document format. However, we introduced this distinction to help the reader at understanding what sort of application domains are involved.

if the set of BibTex files at hand contains a list of equally structured entries, users could easily define a relational schema and write ad-hoc software to move the entries content into the target relational database.

As a further example, consider the HTML files generated as responses of queries run over a remote relational database through a Web interface. As the structure of the query response is presumably fixed in time, hence predictable, one may think of defining a local database to handle the regular information within such files. A programmer may design a database schema corresponding to the identified structure and develop wrappers to translate the HTML pages into data conforming to the schema.

2.2 Schemas

For both data-first and schema first approaches the efficiency of an overall database strictly depends on the relationships between the schema and the database, and the schema and the structure of the information source involved. In the following we capture the various facets of this relationship in the notion of *optimal schema*, and claim that DBMS databases should be used only when built around an optimal schema.

2.2.1 Schema benefits

DBMSs typically provide a *query language* for the specification of *queries*, which are powerful applications for the insertion, modification and retrieval of data into a database. In general, a query can be regarded as a compound of *structural* and *operational* properties. For example, an *SQL* query generally consists of a set of relations (*from* clause) plus a set of predicates over their instances (*where* clause) and a set of operations to be applied over the filtered data (*select*, *insert*, or *delete* clauses). Such queries are performed by traversing only the current instances of the specified relations, selecting the subsets of such instances which respect the given predicates, and eventually returning the result of the operations applied to the data thus filtered.

Query languages do not provide full support for the realisation of complex application systems computing over a DBMS. High-level applications, such as user interfaces, are realised by means of more sophisticated programming languages, whose type systems are compatible with the DBMS data models. Consequently, queries can be perfectly integrated with high-level applications, which otherwise could not interact with the database.

Accordingly, applications over a DBMS may be of various forms and complexity, ranging from simple queries to high-level applications. In this general context, schemas are important in DBMSs for they entail the following benefits:

- system and users have access to a short and neat description of the data *potentially* present in the database:
  - the system can check the *correctness* of the applications: an application is correct if it aims at computing over data described by the schema, i.e. data that can be potentially contained into the database, and, more generally, if run-time errors cannot occur or can be prevented;
  - users have the understanding of what is in the database and can define potentially correct applications;
- the system can optimise both space and time efficiency:
  - databases omit information that may be kept within the schema rather than being repeated in each instance of the data;
  - standard and optimised data access techniques can be devised: being the data stored according to a particular structure, if statistical information is kept up to date, a *query optimizer* can be built, as well as data structures for intelligent access to the data (indexes).

Correctness of applications

Due to the high level of complexity of an application system constructed around a DBMS, the property of *type correctness* is of paramount importance as it ensures that no-run time errors will occur and that no data-inconsistency will arise.

Informally, an application is type correct if the correspondent computations manipulate values only with operations associated with the respective types. This property can be statically checked at compile time, before applications are executed, due to the presence of types assigned to the input and output of the applications, and to the set of pre-defined operators applicable over the values of these types.

However, DBMS applications also interact with queries, i.e. applications specifically operating over the database. Accordingly, so as to ensure that the result of a query can be correctly used by other applications, the query must be proven correct with respect to the database. *Query correctness* should hold whenever a query is executed and can be generally defined as follows:

Definition 2.2.1 *(Query correctness in DBMS) Let Q be a query. Q is* correct *if* there can be *data in the database that satisfy the structural properties of Q.*

This property ensures the *good sense* of Q, which means that if Q yields an empty result this can be interpreted as *no data in the database satisfied the predicates of the query.*

Finally, data is stored in a database according to a precise format, strictly related with the structure of the data as described in the schema. This means that at any

time no datum in the database may have structural properties not described by the schema. Therefore, modulo schema modification, queries can be checked for correctness, once for all, at compile time, by matching the structural properties of the query against the schema.

2.2.2 Optimal schemas

The *quality* of a database depends on the relationship between the schema and the structure of the information source, i.e. a problem domain or an electronic data source, to be handled with a DBMS. Moreover, DBMSs rely on a great deal of system artifacts, whose *performance* strictly depends on the relationship between the schema and the database. A schema that satisfies at best these relationships is called *optimal schema*.

Definition 2.2.2 *(Optimal schema)*
A schema is optimal *in presence of:*

- good modelling*: the static irregularities of the information source to be handled by the DBMS, i.e. the properties of entities which are not common to all entities of a category, are well-described by the schema. This is the case when the relationship between schema and information source is not affected by an excessive use of* information loss *or* typing loss*, where:*
  - *information loss is adopted to produce a simpler schema: the static irregularities of the information source are not represented in the schema;*
  - *typing loss is adopted to produce a homogeneous schema: the static irregularities of the information source are extended to all entities of the category.*
- data efficacy*: the relationship between schema and data fulfills the following requirements:*
  - *the schema is a short description of the data: in presence of a schema as large as the data, optimisation techniques are totally obsolete;*
  - *the schema is quite stable in time: modification to the schema are notoriously expensive in terms of human work, as they require the modification of old data and applications [74].*

DBMSs research studied how to efficiently handle *regular information sources*, which are those leading to optimal schemas. Regular information sources are characterised by a structure which is quite stable in time and is not affected by structural irregularities, i.e. in general, entities belonging to the same category feature the same set of properties.

In the following we study information sources that cannot be efficiently handled by a DBMS, and claim they are the main motivation behind the introduction of SSD research.

2.3 Irregular information sources

Despite of the database design approach adopted, be it schema-first or data first, when the resulting schemas are *not optimal*, traditional database methodologies turn out to be extremely inefficient. This happens for *irregular information sources* whose importance derives from the structural irregularity and instability they feature, i.e. those affected by one or both of the following irregularities:

- *static irregularities*: entities of the information source that belong to the same category feature different properties; the structure of the generic instance of such information source is inherently irregular, thereby good modelling or data efficacy cannot be kept over a reasonable threshold;
- *dynamic irregularities*: the structure of the information sources is unstable and frequently changes in time, hence data efficacy cannot be provided.

Information sources bearing either static or dynamic irregularities are called *irregular*, as their inherent structure does not lead to an optimal schema. However, as we shall see in the next Section, irregular information sources may be handled electronically through SSD technology. Below, we exemplify known schema-first and data-first irregular information sources.

2.3.1 Irregular problem domains

Today, common examples of irregular problem domains are given by problem domains of specific research fields, such as biology, palaeobiology, and similar ones.

For instance, although fossil information sources generally present a common structural pattern that could be represented in a traditional database schema, each fossil may also be associated with further peculiar, relevant information. Moreover, the incessant discoveries constantly introduce new fossils, hence new structural properties.

Such information source is certainly affected by static irregularities as the category of fossils features entities with relevant and different properties. Any attempt to model such scenario with DBMS data models would either make heavy usage of information loss, in order to keep only the properties common to all fossils, or type loss, in order to extend the properties peculiar to each fossil to the category of all fossils.

An alternative is that of considering the category of fossils as a compound of different categories, each peculiar to a limited number of fossils, i.e. those featuring

the same properties. In this case, good modelling may be provided, but data efficacy would be very low, as the schema would be almost the size of the data.

Finally, note that in such scenario, data efficacy would be quite low anyway, due to the frequently changing structure of the information source. Indeed, this would entail changes to the schema, which are likely to imply modifications to both the data and the applications.

For the reasons exemplified above, whenever a problem domain is affected by static and dynamic irregularities, DBMSs cannot be generally adopted and other tools should be employed instead [87].

2.3.2 Irregular data sources

In data-first approaches the structure of relevant data plays an important role as it identifies interesting data in the source documents or database, and it may implicitly define a schema for the target database. However, the resulting schema may not be optimal. Indeed, blending different data sources together and storing them in a DBMS database, or either moving document information into databases, may lead to irregular collections of data. Next, we discuss these common scenarios.

Integration of different information sources

A challenging issue in database research is that of the integration of heterogeneous information sources, in order to query them together in the same database system.

The problem is that of blending data deriving from various information sources, such as relational or object-oriented databases, the Web, file systems, and others, in an integrated data repository, so as to query them all together. Data in such a repository could be modelled in the general framework of object-oriented data, but the overall structure is likely to be irregular. Indeed, some objects may have missing attributes, others may have multiple occurrences of the same attribute, the same attribute may have different types in different objects, semantically related information may be represented differently in various objects. The resulting data is therefore inherently irregular, and cannot be efficiently stored in a DBMS database.

Legible documents

Generally, the structure of information stored within documents is not optimal. Indeed, optimal structure is typically associated with data whose main purpose is that of being queried over, while the purpose of documents is that of being read by humans and interpreted by specific applications.

For example, in BibTex files it is customary to find compulsory entry fields missing. Furthermore, while some fields have meaningful structure, e.g. *author*, there are complex features, such as abbreviations or cross references that are not easy to describe in some database systems (cf. [1]). Due to these static irregularities,

the application of data-first techniques to BibTex files is likely to lead to an inefficient database.

In fact, the structure of data stored in legible documents is often irregular, unknown in advance, and even when it is known, it may change without notice. For such reasons, documents constitute a potential source of irregular information.

Data on the Web and eXtensible Markup Language

The success of the World Wide Web is largely due to the adoption of *HTML* (*Hyper Text Markup Language*) [83], and, more recently, to the introduction of *XML* (*eXtensible Markup Language* [24]). Both markup languages have been proposed as international standards for publishing Web documents, providing a common, simple and human legible format for documents. Since Web-documents favored quick and easy information exchange, people from the Web community started to re-design or convert their data onto Web documents so as to make them available to an increasingly wider community.

In particular, due to its flexibility and expressiveness, nowadays XML incessantly plays the role of a standard *data-exchange* format rather than that of a standard *document-exchange* format. XML documents are explicitly intended as information to be queried over, this fact generating an enormous demand for XML-as-database technology.

For example, consider the Home pages of the academics of a Computer Science Department. These pages may contain some similar information, such as *name*, *e-mail*, *photo*, *age* and *address*. However, some of this information may be missing in some pages, while extra information may be present in others. Transferring this information into a database may result in a quite inefficient setting, due to the bad definition of the corresponding schema. In fact, since all the information should be preserved, the schema would be affected by typing loss or information loss, or lead to a schema that is almost a copy of the data. In addition, once the mapping from the HTML (XML) source onto the database schema is designed and the correspondent wrappers are written, there is no certainty for future HTML (XML) pages to fit with the current schema. Accordingly, HTML documents, as well as XML documents, are often taken as examples of irregular information sources.

2.4 Semistructured Data

We have seen that irregular information sources are characterised by a structure that is either:

- too variable to be represented by a stable schema;
- too irregular to be represented by a short and clear schema.

These observations led to the definition of *semistructured data models*, according to which databases are self-describing collections of data represented as rooted, directed, labelled trees or graphs [25]. Note that, unlike traditional data models, which provide a set of abstractions to be appropriately combined to define the conceptual model of a database, SSD models offer just one abstraction mechanism. Accordingly, system developers are not concerned with the creation of a schema, which is implicitly defined by the data model and describes all possible *semistructured databases* (*SSDBs*) as labelled trees or graphs. In this work we define *semistructured data* as data stored in an SSDB.

The most representative semistructured data model is the *Object Exchange Model* (OEM) [81]. The novelty of this data model is in the fact that it implicitly defines a schema for all possible databases, which are represented as rooted, labelled, directed graphs with values into the leaves. In particular, OEM consists only of the definition of the set of SSDBs by means of the following BNF:

$$
\begin{aligned}
db \quad &::= Node \\
Node \; &::= < id, Label, Value > \\
Value &::= Atomic \mid \{Node, \ldots, Node\} \mid id
\end{aligned}
$$

where the *identifiers id* have unique identity in the model.

From the modelling point of view, unlike traditional data models, OEM provides a conceptual model with the unique category of graphs and, by the grammar above, implicitly defines the corresponding schema. Thus, the representation of the structure of an information source is left to the expressivity of the single database, which can only represent entities and properties, i.e. associations, between entities.

By definition, entities of an instance, which are represented by identifiers in the database, are possibly related with other entities for being their property values; properties are uniquely represented in the database by pairs $(label, id)$, where id is a target node. Note that identifiers are considered as independent individuals and not as members of a class or a relation as in traditional data models.

Figure 2.3 graphically exemplifies an OEM collection. Each identifier is associated with a label which provides its description, and may have an arbitrary number of children that may be equally named. Furthermore, a child may be an ancestor of its parent.

Note how the data model consents the definition of databases featuring any sort of static irregularity. Entities that belong to the same category, such as the entities labelled as *Professor* in the example, may be feature different properties, e.g. *Phone*. Furthermore, there is no restriction on the names of the properties of an entity, which can be repeated as for *Professor* for the entity named as *Fibonacci*. Finally, other entities, with arbitrary structural properties, may be added to the actual database at any time.

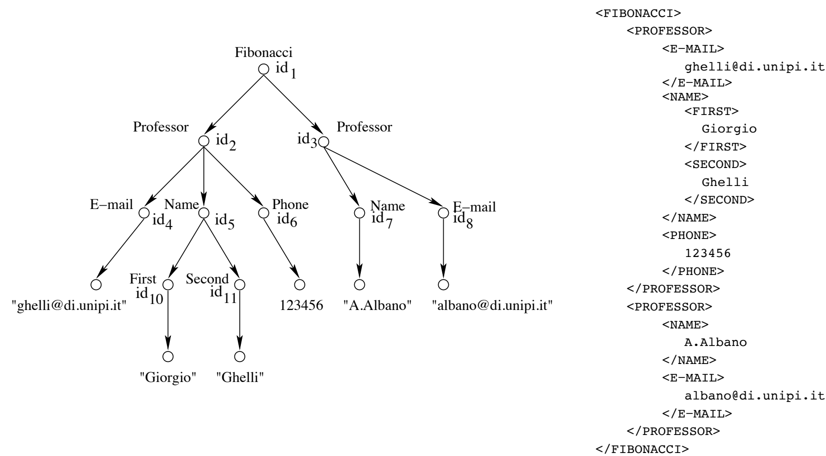

Figure 2.3: OEM and XML SSDBs of the Fibonacci's group pages

Different authors considered specific restrictions or representations for their data models, depending on the issues they were seeking. Typical examples are, narrowing the analysis to trees, so as to avoid tedious reasoning about cycles and shared nodes, and considering labelled edges rather than labelled nodes. A notable case is that of XML documents, which are a widely accepted representation for SSDBs due to the hierarchical structure of the format (cf. [4, 88]). For instance, consider the XML document corresponding to the OEM collection in Figure 2.3.

An SSDB is a representation of the structure and of the instance of an information source. This integration makes SSD technology extremely flexible as it gives to the user the ability of:

- freely inserting or deleting data representing portions of the information source at any time;
- modifying the database in correspondence of changes to the information source at any time with very low costs.

In practice, these models have been applied to quite a few research prototypes, working on the areas of data integration and conversion (cf. [36, 39, 45]), Web Site management (cf. [86, 54, 59]), general purpose management of semistructured data (cf. [29, 51]), and XML data management, which we shall discuss in the following.

2.5 Semistructured data manipulation

SSD models provide the flexibility required to represent SSD. On the other hand, system and users are provided only with:

1. the semantic information given by the labels;
2. the implicit structural information derived from the hierarchical structure of the database.

Consequently, the data can only be accessed and manipulated as a labelled graph with unknown topology. These features considerably limit the range of possible operations over the data, which can only be treated as graphs, and inevitably reduces the number of system facilities, such as query optimisation and the usage of indexes over the data. In this Section we introduce the general feature of the most common semistructured query languages.

2.5.1 Query languages

Semistructured Query Languages (*SSDQLs*) are fundamentally tools to identify possibly more regular, smaller, views of large SSDBs. Views are the result of the execution of queries, which specify a set of assumptions over the structure of the data to be queried.

For instance, a query over the collection in Figure 2.3, written in *Lorel* [3], the language defined by the *TSIMMIS* project group, looks like:

```
select result:x.Second
from Fibonacci.*.Professor x
where ''ghelli'' in x.E-mail
```

The query searches the surname of those professors whose e-mail contains the string *ghelli*, and returns a tree-structured SSDB with one edge *result* for each of the surnames found in the process. Moreover, even though this does not apply to our example, the clause $*$ states that the node labelled *Professor* could have been at arbitrary depth in the database graph, as long as it was a descendant of the root node labelled *Fibonacci*.

An SSDQL query consists of three parts, to be executed separately to get a result: *binding*, *filtering*, and *constructing*.

First a set of candidate nodes is identified, by providing the structure through which they must be reached from the root of the graph, i.e. the entry point of the database. Subsequently, the resulting set is bound with a variable, such as x in the sample query above.

The structure for the identification of the candidate nodes is generally given in terms of a set of *paths* of the form $l_1, l_2, \ldots, l_n$ to be matched against the SSDB.

The result of applying such paths to an SSDB graph, is the set of nodes e_n such that there exist an edge $(r, l_1, e_1), (e_1, l_2, e_2), \ldots, (e_{n-1}, l_n, e_n)$ in the SSDB, where r is the entry point of the database.

A query specifies a set of paths by means of *Generalised Path Expressions* (*GPEs*, cf. [38, 40]), which are expressions of the form $p_1.p_2, \ldots .p_n$, where the p_i's may be a label l or particular symbols, known as *wild cards*, of the form $*$, ?, |. For example, the GPE $l_1, l_2, \ldots, l_n$ represents only the relative path, while the GPE $l_1. * .l_2$ represents the infinite set of paths which begin with a label l_1 and reach a label l_2 after encountering an arbitrary number of labels.

The process of querying an SSDB is based on pattern matching techniques exploiting the inherent graph structure of the given data model and the semantic information coded by the GPEs.

Afterwards, the set of nodes can be filtered through a set of predicates applied over the corresponding binding variables, such as `''ghelli'' in x.E-mail` in the query above. The set of nodes triggered by the predicates is in turn applied to a set of constructors, such as `result: x.Second`, which return an SSDB as a result of the query.

The expressive power of SSDQLs is measured on the base of the kind of queries the user can express (cf. [28]). Most languages provide the basic operations required by relational query languages, such as joins and grouping, plus some form of restructuring, i.e. the ability of creating a new SSDB from another one.

Well known query languages are *Lorel* [3, 82] developed by the *TSIMMIS* group, *StruQL* [55], *UnQL* [29], developed by the UnQL group, *YATL* [39] developed by the Verso group, *TQL* [33], and others. These languages typically provide SQL-like constructs, through which programmers may create and query SSDBs.

Furthermore, in the last few years the *World Wide Web Consortium* (*W3C*) focused on the design of standard data models [56] and query algebras [53] for XML SSDBs, so as to provide general guidelines for commercial developers. Well known XML query languages are *Lorel* adapted to XML [63], *XQL* [84], and *XML-QL* [49], *XPATH* [40] developed by the W3C, *XDuce* [66], *Quilt* [35], and *XQuery* [34, 70]. In the next Chapter, we shall see that *XDuce*, *Quilt* and *XQuery* support types for SSDBs, and are the only example of SSDQLs with the expressive power of Turing-complete languages.

2.5.2 Query languages implementation

The implementation of SSDBMSs is a known and interesting problem, which embraces most of the issues of traditional database design plus others, stricly peculiar to the challenge of querying SSDBs.

Two main approaches have been explored. The first, and most researched, approach is based on the storage of data in a relational, or object-relational, DBMS, and on the translation of the queries into SQL queries [61]. *Quilt* [32] and *XML-QL*

have been implemented in this way. These implementations usually are not able to support all operations one would require in an SSDQL, due to limitations of the underlying DBMSs which are definitely intendend for different purposes [85].

Another approach, which is much more expensive but overcomes these limitations, consists in the construction of a new complete and specific system. For example, *Lorel* [71], *StruQL* [50], *XML-QL*, and *XQL* have been implemented in this way, while the implementation of *Xyleme* [9] is somehow intermediate. This approach requires the definition of a new storage model [67], a specialized query algebra, as well as the definition of adequate cost-based optimisation techniques. While some efforts have been spent in the context of *Lorel* and, to some extent, of *StruQL*, satisfactory algebras and general cost models for such implementations are still missing.

2.6 Semistructured data drawbacks

So far, we have shown how the choice of SSD models may overcome the problem of using otherwise inefficient DBMSs to manage irregular information sources. On the other hand, SSDBMSs are not as powerful as DBMSs as explained in the following.

- *Query optimisations and data access cannot be supported*: due to the absence of a schema the only possible execution plan is the exhaustive traversal of the whole database according to the structure specified by the GPEs; furthermore, the data cannot be stored according to particular data structures, exploiting indexes as in DBMSs, but only according to strategies for the memorisation of labelled graphs.

- *Query correctness cannot be supported*: with reference to Definition 2.2.1 of query correctness for DBMSs, in SSDBMSs query correctness cannot be statically checked as there is no notion of static schema. Indeed, as structure of the database may change at any time, query correctness should be checked whenever a query is executed. Furthermore, since in SSDBs the structure of the data is intermixed with the data, query correctness should be matched against the actual SSDB.

  In conclusion, query correctness in SSDQLs may be dynamically checked, during query execution, by verifying that all paths specified by the query found a match in the traversal of the SSDB. This way, when a query returns an empty result the user learns if this was due to the erroneous GPEs he had specified or, more interestingly, because none of the candidate nodes identified in the binding part survived the query filtering.

  Clearly, this notion of correctness cannot be a discriminating factor for the repeated execution of the query as it could be in a DBMS. Indeed, a query may turn out to be incorrect for one execution and, due to modifications to

the database, correct for the one immediately after. Furthermore, due to the size of a database and to its potentially changing topology, users cannot be aware of the precise structure of the database.

In fact, users define queries after an eye-inspection of the database content, so as to identify peculiar structural properties, and queries are performed in a *greedy* fashion, in an attempt to get back as much information as possible.

- *The language type system is poor*: the absence of a schema and, consequently, of a static form of query correctness, hampers the integration of SSDQLs with traditional typed languages, which require the specification of types for the values to be manipulated and whose type systems are usually much more sophisticated than semistructured data models. Accordingly, the level of complexity of SSDBMS applications is generally limited to the SQL-like queries presented above. Computations over the database may modify the content of an SSDB only by renaming some edges or nodes, changing values in the leaves, or adding nodes and edges to the SSDB.

- *Queries may be hard to define*: the absence of schema, hence of a short description of the data, typically hampers the definition of queries looking for specific data in the SSDB. The users may get hold of structural information only after an eye-inspection of higly-irregular graph-like databases.

There are two main trends in SSDQLs realisation. Those that do not take into account any form of meta-information and those that are based on the assumption that the SSDB comes with a schema. The former are usually a specific case of the latter when a schema is not available. In the following chapter we discuss the setbacks listed above and show how some of them have been overcome by reintroducing the concept of type for SSDBs in SSDQLs.

Chapter 3

Typing semistructured data

In the previous Chapter we have highlighted the differences between SSDBMSs and DBMSs. In particular, we concluded that SSDBMSs can be applied to handle any kind of information sources, while DBMSs offer notably efficient services when applied to information sources with stable and regular structure and show very poor performances when applied to irregular information sources. Consequently, for strongly irregular information sources, SSD technology seems to be the only possible solution. However, the lack of schema generates a notable distance between SSDBMSs and DBMSs, summarised by the following aspects:

- *application correctness*: correctness of applications over a database cannot be guaranteed, while in DBMSs it is verified at compile time by means of type checking mechanisms;
- *operations applicable over the data*: data manipulation is limited to queries upon a labelled graph structure, while in DBMSs it is open to sophisticated computations over various typed data structures, such as relation, records, collections, and so on;
- *system optimisations*: queries may be executed by simply sequentially traversing the whole database, while in DBMSs queries are performed by exploiting various forms of optimisations for query execution and data access;
- *user knowledge*: users may come to know the content of a database by an inspection of the overall database, while in DBMS users have access to a schema, which is a short description of the database's content.

Nevertheless, structure-less is not a proper definition for semistructured data models. Indeed, each datum in an SSDB is reachable by traversing the database according to a specific *structure*, i.e. a sequence of labels from the entry point, called *path*. In this Chapter we discuss when and how a separate description of such structure, i.e. a schema, may be statically or dynamically provided for SSDBs.

In particular, we shall see how the choice of either typing methodology for SSD is concerned with a general trade off between limiting the flexibility of SSD models and improving all of the benefits deriving from meta-information.

3.1 Static and dynamic typing for SSD

The observation that data in SSDBs are identified by a precise structure has motivated a number of investigations into the applicability of typing techniques to SSDBMSs. There are essentially two different research trends aiming at typing SSD, according to which a type may be provided statically or dynamically.

Static typing The relationship between types and data is that of conventional DBMSs, i.e. the data in the database must structurally *conform* to the schema. We shall see that different definitions of schema and conformity may be given, so as to capture irregular data and guarantee some of the benefits of static typing at the same time.

Dynamic typing This approach does not impose any constraint on the structure of SSDBs, which can be arbitrarily organised; readable and possibly succinct forms of meta-information are inferred from the structural information represented by the labels of SSDBs, in order to be exploited by the user to better define his queries and by the system to support forms of query optimisation.

In the following we present static and dynamic approaches together with some examples of well-known approaches.

3.1.1 Static typing

Static typing techniques for SSD mirror DBMSs schema-first methodologies in that a schema is provided before data population. In both scenarios, this requirement entails a strong assumption of *structural stability* of the information source at hand, whose structure must be fixed in time and known in advance. In general such assumption excludes the applications of such techniques to irregular information sources affected by dynamic irregularities, and static typing techniques are customary applied to information sources mainly affected by static irregularities.

The ability of a type language to describe data irregularities is in a trade-off with static typing benefits, i.e. the ability of the system to provide optimisations, effective definitions of query correctness, user knowledge, and sophisticated data structures. Therefore, various forms of static typing techniques for irregular information sources have been proposed, whose nature depends on the amount of irregularities to be handled by the system. So as to reveal this trade off, next we present a well known approach relying on DBMSs systems endowed with peculiar typing methods, which privileges static typing benefits to system flexibility, and some typing approaches for SSD, which gain further flexibility by loosing some of the benefits.

- *DBMS languages with union types*: mildly irregular information sources are represented as values of traditional data models endowed with union types:
  - query and applications can operate on traditional and intuitive type structures, such as relation, records, collections and others;
  - all static typing benefits can be supported.

  DBMSs and programming languages advantages can be exploited, but the amount of irregularities must be very limited due to system efficiency issues implied by union types.

- *SSDBMSs with low-level type systems*: irregular information sources are represented as SSDBs, whose content is described through an algebra of *low-level types*, which typically describe the structure of labelled graphs, rather than that of values such as records and relations:
  - applications are limited to queries operating over labelled graph structures;
  - depending on the relation of conformity between types and SSDBs, query correctness may assume different shapes or not be supported at all, while query and storage optimisation can be generally supported.

  Some benefits of DBMSs static typing cannot be supported, but low-level types and SSDQLs can generally better cope with irregularities than DBMSs with union types. As examples of low-level types for SSD, we shall show some known examples of XML typed query languages, and the peculiar typing of *UnQL* by means of *graph schemas*.

Union types

Untagged union types are introduced in programming languages to increase application flexibility by allowing values of different types to be described by the same one. Informally, the semantics of union types states that a value v conforms to a union type $union(T_1, \dots, T_n)$ if there exists $i : 1, \dots, n$ such that v conforms to T_i. Due to their peculiar nature, union type values should be *projected* into their specific type of the union before they can be referred and manipulated. To this aim, programming languages support particular commands, such as,

```
typeof x is union(T1, ..., Tn)
⋮
union case x of
T1 :< operating on x as a value of type T1 >;
⋮
```

T_n :< operating on x as a value of type T_n >;
endcase;

At run-time the `case` command matches the actual type of the value associated to the variable `x` with the $\mathtt{T}_i$'s and executes the branch corresponding to the matching type. Note that, despite of the dynamic type checking required, the introduction of union types does not compromise static type checking of applications.

Sophie Cluet, in [41], observed that language type systems or traditional database data models, endowed with union types, are apt to describe some mild form of irregular information sources. To be convinced of this, recall that static irregularities derive from entities, represented by values v, each featuring a different set of properties, represented by the types T_i, which all belong to the same category. Union types can be used to represent such category as a collection of type $set(union(T_1, \ldots, T_n))$.

For example, in Figure 3.1, by means of intuitive languages for the definition of values and high-level types in a DBMS application, we provide the value `x` corresponding to the SSDB in Figure 2.3 with the corresponding high-level type `Professors`. Applications and queries operating over the elements of the collection `x` can be checked for correctness, and system and users can take advantage of all benefits usually enforced by DBMSs.

```
let profname = [First = ''Giorgio'', Second = ''Ghelli'' ]
let gg = [ Name = profname, E-mail = ''ghelli@di.unipi.it'',
           Phone = 123456 ]
let aa = [ Name = ''A. Albano'', E-mail = ''albano@di.unipi.it'' ]
let x = { aa, gg }

type Professors = set(Professor)
type Professor = union(record(E-mail:  string;
                              Name:record(First:  string;
                                          Second:  string);
                              Phone:  integer );
                       record(Name:  string; E-mail:string ); );
```

Figure 3.1: Type description for the value representing the Fibonacci's SSDB

Note how a set of OEM nodes with the same label and reachable from the same node, such as `Professor` with `Fibonacci` in our example, intuitively maps into a collection value, such as a relation in relational databases. Similarly, a node reaching differently labelled nodes intuitively maps into a record value.

This approach should be used when the number of static irregularities runs under a certain threshold and the overall data structure is quite stable in time. In

summary, the information source should be *almost* regular. Indeed, a type which accurately describes a value representing a significantly irregular information source would provide little useful abstraction over the database and compromise system's efficiency: such a type would be beyond human understanding and require expensive static and dynamic type checking controls, therefore be of no practical use.

Integration of different SSDBs

A less obvious approach to the typing of irregular information sources has been proposed in the specific field of integration of different databases. This integration process falls in the category of DBMS data-first approaches and requires the recasting of the data sources according to a new unifying schema. However, as explained in Section 2.3.2, the integration of databases representing different information sources is likely to lead to a collection of irregular data. Such data cannot be re-cast under an optimal schema and should therefore be represented as SSD, thereby losing any static type information associated with the original structured data source.

However, sometimes databases are unified because they contain related information which should be merged to be queried over as a single repository, e.g. databases of professors from different departments. Still, although the databases may be described by very similar schemas, as long as the conceptual models were developed by distinct individuals, the resulting data may present static irregularities. In the presence of a limited number of irregularities, the resulting database could be represented by DBMS languages endowed with union types, as shown in the previous paragraph. However, applications written for the original schemas should be rewritten as they might not be correct for the new schema.

Buneman and Pierce [30] claimed that the data arising from this specific form of integration can be represented by SSDBs whose content is described by a particular type system endowed with untagged union types. Union types capture the irregularities of the resulting data, while specific type rules consent to reduce the number of modifications to the applications which are typically required in correspondence with changes to the schema.

The type system proposed by the authors is the following:

```
T ::= T1 × T2 | (records)
      T1 + T2 | (untagged unions)
      set(T)  | (sets)
      l:T       (singletons)
```

which describes SSDBs as trees, represented by nested record values, where the label of the fields cannot be repeated. In particular, record types are defined as products of singletons of the form `l:T`, but may also include elements of the form $T_1 + T_2$. Hence, a further peculiarity of this language are typed operators to access record structures such as $T_1 \times (T_2 + T_3)$, not to be described here.

Databases could be represented by means of such values, while the schemas could find a match, at least a partial match, in the type system above. Thus, SSDBs could preserve, at least partially, some of the static information provided by the schema of the original data source. Consequently, typed applications may be written to operate over the SSDB, potentially exploiting all benefits of static typing.

The type system is provided with type equivalence rules, subtyping rules and a *distribution rule* over record and union types, which defines the following type equivalence:

$$T_1 \times (T_2 + T_3) \equiv (T_1 \times T_2) + (T_1 \times T_3)$$

Assume the SSDB is typed as set($T_1 \times T_2$) and that a new data source, whose schema maps onto a type set($T_1 \times T_3$), should be added to it. Their integration could be typed as set(($T_1 \times T_2$) + ($T_1 \times T_3$)). The distribution rule states that those applications written to operate on the part common to both data sources, i.e. the type T_1, are still correct and can therefore operate on the resulting SSDBs without being modified.

For example consider the integration of two databases constructed to model professors according to the types `db_one` and `db_two`:

```
db_one:
set((Name:  [(First:string) × (Second:string)]) ×
    (E-mail:  string) ×
    (Phone:  integer) )

db_two:
set((Name:  string) ×
    (E-mail:  string) )
```

The resulting database could be typed as `db`, i.e. a set of the union of the core types of the two original sets:

```
db:
set([ (Name:  [(First:string) × (Second:string)]) ×
      (E-mail:  string) ×
      (Phone:  integer) ] +
    [ (Name:  string) ×
      (E-mail:  string) ] )
```

According to the distribution rule, the following type equivalence holds:

```
db = set((E-mail:  string) ×
         [(Name:  (First:string) × (Second:string) ) ×
          (Phone:  integer)
```

```
 +
 (Name:  string)
])
```

Thus, applications operating only on the field `E-mail` and proved correct with respect to both `db_one` and `db_two` can be reused on the new database. In a way, the type system degrades gracefully when new data sources, with variation in structure, are added to the SSDB, while preserving the common structure of the data sources where it exists.

The authors claim that for the amount of irregularities generally encountered in this particular form of data integration, the system shows good performances; in particular, the complexity of the distribution rule algorithm is reasonable. The ideas behind this type system have been extended in subsequent works, in particular in the realisation of the XML processing functional language *XDuce* [66].

Types for SSDQLs

SSDQLs should be adopted to handle information sources whose irregularities could not be efficiently handled by DBMSs. When dealing with SSDBs, types are abstractions as expressive as GPEs over labelled graphs, and conformity is a matching relation between the structure provided by the type and the structure of the actual database: a database conforms to a type if the paths defined by the type are present in the database according to the type semantics.

For instance, consider XML query languages [57]. The spread of standards for specifying XML meta-information, such as *DTDs* [15] at first and *XML Schema* [52, 89, 21] afterward, called out for the realisation of typed XML query languages, capable of exploiting a schema if available. DTDs and XML schemas are low-level type systems, describing the nested, tagged structure of XML documents and capturing the static irregularities of the data.

For example, a DTD for the XML document in Figure 2.3 may look like:

```
<!ELEMENT Fibonacci (Professor*)>
<!ELEMENT Professor (Name, E-mail, Phone?)>
<!ELEMENT Name ((First, Second) | ♯PCDATA)>
<!ELEMENT E-mail ♯PCDATA>
<!ELEMENT Phone ♯PCDATA>
```

This schema requires the XML documents conforming to it to have an entry point labelled as `Fibonacci`, nesting an arbitrarily long sequence of tags `Professor`. Furthermore, each of such tags should feature two tags `Name` and `E-mail`, possibly followed by an optional tag `Phone`. Finally, `Name` may nest either a simple string, denoted by the type `♯PCDATA`, or a sequence of two tags `First` and `Second`, in turn of type `♯PCDATA`.

Typed XML languages range from those capable of querying typed data to those providing Turing-complete programming paradigms. Examples of the first kind are *XSL* [91], *YATL*, *TeQuyLa* [10, 12, 11], while examples of the second kind are *XDuce*, *XQuery*, and *Quilt*.

For instance, *XDuce* is a Turing-complete typed programming language for the definition and manipulation of XML documents. The type system, which is based on the ideas exposed in [31] and reported in the previous paragraph, have the same expressive power as DTDs and describe XML documents. Language programs are second order typed functions operating over the data that can be checked for correctness before execution. Interestingly, types are used as a matching tool for the data at run-time: data dereference is performed by matching types with data at run-time, in a way that resembles that of untagged union type values. This is because the generic XML document may be described by different types and accessed according to different interpretations.

Based on types, some optimisation techniques have been designed, but not yet developed, for nested queries [44, 42, 68] and GPEs matching [73, 72, 38, 60]. Due to the variety of application contexts, however, there are no precise definitions of query correctness. Correctness is generally intended as a relation between the structure specified by the GPEs in a query and the schema of the database. For example, consider again the XML document in Figure 2.3 and the DTD given above. The query

```
select x
from Fibonacci.Professor.(Name | Fullname) x
```

searches for paths from the root of the document that match either the structure `Fibonacci.Professor.Name` or `Fibonacci.Professor.Fullname`. The query may be considered as correct or incorrect, depending on the kind of correctness policy adopted.

Existential approaches to correctness would establish that the query is correct because there exists a non-empty intersection between the paths defined by the schema semantics and the paths specified by the query semantics. The rational underneath such policy is that a query is correct as long as there is a chance to produce a result. The user may be possibly warned by the system about the fact that his query is searching for paths that cannot be found in the database. Such policy, which is inappropriate for traditional type checking, is quite reasonable when dealing with SSDBs.

Universal approaches to correctness would instead establish that the query is not correct because the query searches for paths not described by the schema. The underlying policy may state that a query is correct only if all paths in the query find a match in the schema; in other cases, the policy may be more restrictive, stating that a schema with a union type can only be accessed by queries which specify a

case for each member of the union, i.e. the query must match at least one path in the database.

Other refinements of these definitions are possible, each leading to a different notion of query correctness. In general, due to the variety of application scenarios, it is hard to find a solution which applies to all situations. Indeed, only few languages provide a definition of query correctness, with the notable exception of *XDuce*.

The drawbacks of these approaches are that applications are quite poor, as they operate on data modelled as an XML graph; furthermore, as pointed out in the introduction of the Section, these techniques are pointless in presence of strongly irregular information sources.

UnQL and Graph schemas

In the language *UnQL* [29], SSDBs are edge-labelled graphs where values are specified as the last edges of paths. For example, the *UnQL* database corresponding to the OEM collection in Figure 2.3 is illustrated in Figure 3.2.

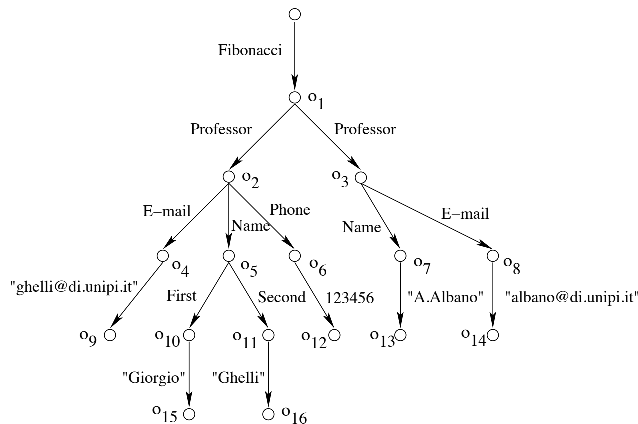

Figure 3.2: UnQL database

In a later stage of development, the language, which is similar to the SSDQLs presented in the previous Chapter, has been extended with a type system [26, 60, 58]. In *UnQL* a type is a graph, namely a *graph schema*, whose edges are marked with constraints, namely unary formulas, over the domain of SSDB labels. A SSDB *db* conforms to a graph schema G ($db \preceq G$) if it is *similar* to it. Informally, this means

there exist a relation $\mathcal{R}$ between the nodes of the SSDB and the nodes of the schema such that:

1. the roots are in $\mathcal{R}$,
2. $(o_1, o'_1) \in \mathcal{R}$ if for each edge (o_1, l, o_2) outgoing a node o_1 in the SSDB there exists an edge $(o'_1, p(x), o'_2)$ outgoing the node o'_1 of the schema such that:
   (a) the label associated with the former edge verifies the predicate associated with the latter edge: $p(l)$ is true;
   (b) o_2 and o'_2 are in $\mathcal{R}$.

Note that, graph schemas are not as restrictive as traditional programming language type and low-level types, in that conformity does not enforce the presence of a label outgoing a node. As a consequence of this, the empty SSDB conforms to all possible graph schemas.

Figure 3.3 illustrates an example of graph schema for the database graph in Figure 3.2. Note that, for simplicity, the predicate $p(x) \equiv (x = \texttt{Fibonacci})$ is simply replaced with `Fibonacci`. G_1 states that if there is an edge labelled *Fibonacci* outgoing the entry point of the database, such edge reaches nodes which have an outgoing edge labelled as *Phone* or others which are not equal to it. In turn, edges labelled *Phone* lead to nodes whose outgoing edges can only bear integer values.

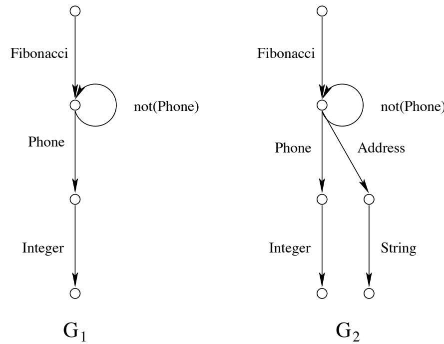

Figure 3.3: Graph schemas

Query correctness cannot be supported because the notion of schema is too loose and does not provide complete information about the data. For instance, the

database graph in Figure 2.3 conforms also to G_2. A query Q selecting entities reachable with the path $Professor.Address.x$ should be intuitively correct with respect to G_2. However, its result would be empty because the data cannot satisfy the structural properties of the query.

In practice, *UnQL* exploits graph schemas for the definition of typed views over the data (G_Q's), to provide user knowledge, and to improve query optimisation and decomposition.

3.1.2 Dynamic typing

Dynamic typing approaches exploit typing information mostly with the purpose of enabling resource optimisation and promoting user understanding of the data. Given an existing SSDB, information about its structure is automatically or semi-automatically inferred by the system.

Notable examples of dynamic typing methodologies are *Representative Objects*, *Dataguides*, and *approximate types*, which we shall discuss in the following.

Minimal type inference

These techniques are concerned with inferring a detailed and minimal schema from a given SSDB. *Dataguides* [90, 64] and *Representative Objects* [78] are examples of such schemas.

For example, representative objects are labelled graphs obtained from the original SSDB by keeping track of all possible paths from the root. One such schema is the minimal description of an SSDB and can be used for schema browsing, user knowledge purposes, and basic forms of query optimisations, such as avoiding the execution of queries whose result is proven to be empty. Techniques for schema maintenance have also been developed.

In the presence of SSDBs with shared nodes nested at various levels, schema inference becomes very expensive. These methodologies, however, are also doomed to fail also in presence of a sheer number of irregularities. Indeed, the size of the resulting schema would not provide useful information to the user.

Approximate typing

Minimal types provide useful but limited abstraction over an SSDB. In particular, when the data is extremely irregular, the number of possible paths in the SSDB becomes overwhelming and defining precise queries turns into a challenging task. This motivated the development of more sophisticated type inference techniques [77, 76], whose main aim is that of identifying some regular structure underlying very large SSDBs. Nodes in the SSDB are assigned a type, while trying to minimise the *total number of types* and the *deficit*. Deficit measures, given an association between

a node and a type, the amount of information related to the node, typically its outgoing and/or incoming labelled edges, which is disregarded by the type.

Informally, the methodology is based on (*i*) a distance function between SSDBs and (*ii*) a measure for the size of types. Given an SSDB *db*, the problem is that of finding a set of types τ and a database db' such that (*i*) db' is typed as τ, (*ii*) the size of τ is smaller of a given threshold, and (*iii*) the distance between *db* and db' is minimised. The resulting type, together with the associations of nodes to the corresponding types, may then be passed to the user, for user knowledge purposes, and to an optimiser, in order to improve query evaluation.

The authors proved the problem of inferring a perfect type for an SSDB to be NP-complete, and their solution relies on heuristics capable of calculating the *best approximate type* for an arbitrary database. They developed two different strategies, the first based on schemas as Datalog programs, and the second resulting in a traditional structure of nested record types. The difference between the two rests in the notation used: monadic Datalog programs describe SSDB by defining classes of nodes in terms of their incoming and outgoing edges, while records characterise a sets of nodes in terms of their outgoing edges, here corresponding to record value fields.