A FOUNDATION FOR INTEGRATING HETEROGENEOUS DATA SOURCES

NARAYANA IYER SUBRAMANIAN

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

AUGUST 1997

0-612-25935-8

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Narayana Iyer Subramanian**
Entitled: **A Foundation for Integrating Heterogeneous Data Sources**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________ Chair
______________________ External Examiner
______________________ Examiner
______________________ Examiner
______________________ Examiner
______________________ Supervisor
______________________ Co-supervisor

Approved ______________________
Chair of Department or Graduate Program Director

__________ 19 ____ ______________________
Dean of Faculty

Abstract

A Foundation for Integrating Heterogeneous Data Sources

Narayana Iyer Subramanian, Ph.D.
Concordia University, 1997

We study the foundations of the integration issues that arise in a federation of heterogeneous data sources, possibly storing related information. Some of the notable features of our approach, motivated by the shortcomings of existing technology, include *(a)* the ability to share data across multiple heterogeneous data sources, *(b)* the ability to manipulate the meta-data (schema) component of a data source in the same vein as data can be manipulated, and *(c)* the ability to query besides well-structured data sources (such as relational databases), semi-structured data sources (such as the HTML documents on the World Wide Web). Our approach is declarative and is based on a simple logic called *SchemaLog*. We initially develop SchemaLog in the context of multiple relational databases. SchemaLog's syntax is higher-order but it enjoys a first-order semantics. We present a formal account of the semantics of SchemaLog by developing a model theory, a proof theory, and a fixpoint theory. We also prove that these semantics are equivalent for SchemaLog. We illustrate the simplicity and power of SchemaLog with a variety of applications including multidatabase interoperability, database programming with schema browsing, database restructuring, cooperative query answering, and powerful forms of aggregate computations, in the spirit of OLAP applications.

SchemaLog can be implemented on top of existing database systems in a 'non-intrusive' way. We describe such an implementation architecture. Realizing an efficient implementation of a SchemaLog-based system warrants the study of the calculus and algebraic languages underlying SchemaLog. We develop a new algebra by extending the conventional relational algebra with some new operations that are capable of manipulating both data and schema information in a federation of databases. In particular, the algebra has schema querying as well as schema restructuring operations.

We also develop a calculus language inspired by SchemaLog. Based on the calculus language, we study varying notions of safety that naturally arise in a federation scenario.

One of our primary concerns in this dissertation has been the practical relevance and industrial impact of our contributions. In this vein and inspired by the SchemaLog experience, we develop a principled extension of SQL, called *SchemaSQL* . *SchemaSQL* is downward compatible with SQL syntax and semantics and is capable of *(a)* representing data in a database, in a structure substantially different from the original database, in which data and meta-data may be interchanged, *(b)* creating views whose schema is dynamically dependent on the input database, *(c)* expressing novel aggregation (over rows, and in general blocks of information) operations, in the spirit of some of the functionalities needed in OLAP applications, and *(d)* providing a great facility for interoperability and data, meta-data management in multidatabase systems. We also describe an architecture for realizing a *SchemaSQL* implementation that makes use of existing technology.

Legacy as well as non-traditional information systems constitute an important fragment of the data sources available in real-life. We demonstrate that SchemaLog can be naturally extended to support non-relational systems as well. In particular, we show that the powerful features of SchemaLog for meta-data querying, information restructuring, and interoperability extend naturally to the ER databases, Network databases, and information sources on the web. In particular, we address the fundamental problem of retrieving specific information of interest to the user, from the enormous number of resources that are available on the Web. With this in mind, and inspired by SchemaLog, we develop a simple logic called *WebLog* and illustrate the simplicity and power of *WebLog* for Web querying and restructuring using a variety of applications involving real-life information in the Web.

To My Parents Narayana Iyer and Lalithambika,
Brothers Ganesh and Girish, and
Sister Radhika

For their love, compared to which the oceans that separate us are little drops

Acknowledgments

Friend, Philosopher, and Guide might be a cliche, but no other phrase better captures the role Dr. Laks V.S. Lakshmanan, my principal advisor played in my five years at Concordia. When I started my Ph.D. with him as a virtual infant in the area of logic and databases, he, with his infinite patience and loving care, helped me learn and grow to a stage, I never imagined I would reach. I will always be indebted to him for making me believe in myself, for helping me dream, and for teaching me how to make dreams come true. I am sure to reminisce every moment of the times we spent together in his office, the conference rooms, the luncheon room, the air planes, and the Al Van Houttes and other coffee joints in Montreal, chatting on music, movies, culture, and most aspects of life, besides our research. He has been an ideal supervisor to me, always available whenever I needed him - be it weekends or late nights, and I am ever indebted to him for that. Doing research with Laks has always been a delightful experience. His constant encouragement and inspiration, and the contagious zest for life he exuberates, have had an everlasting effect on me. My heartfelt thanks to him for all that he has done for me.

My other advisor Dr. Fereidoon Sadri also provided me exceptional guidance, and has been a constant source of support and inspiration during my stay at Concordia. I vividly remember the many meetings I had with him especially in the early stages of my research - Fereidoon was always encouraging and had a knack of making me feel good at the end of the meeting, no matter how ridiculous my ideas were! I am happy that his optimism has finally paid off. Despite his absence from Concordia in the last two years of my research, he was always in touch with me via e-mail and never missed opportunities such as conferences and visits to Concordia, to meet and advise me on my research. I thank him also for the faith he had in me and for all his encouragement. I will ever remember the summer of 94 when we both played tourists roaming around the nook and corners of Quebec.

Dr. V.S. Alagar was instrumental in having me join Concordia. He, as well as Dr. T. Radhakrishnan have in various ways made my stay in Concordia, a very pleasant one. I am grateful to them for their encouragement and appreciation.

I would also like to thank Marc Gyssens of University of Limburg, with whom I had the fortune of working on issues related to querying and restructuring tabular information. I have learnt quite a lot from him and will always be indebted to him for that. The fun filled meeting I have had with him and Laks will ever be green in my memories (and hopefully would not *blow up my brain*!)

Laks' wife, Sharada, took tremendous interest in my well-being. She has also cooked many wonderful dinners for me. I thank her for these. I also thank Laks' daughter Kaavya for her love for 'Sububu' and for putting up with the many technical discussions Laks and I used to have in her presence.

Alanoly Andrews, Hasan Jamil, and Nemat Shiri made my life fun in the office. I will ever remember the debates on issue such as *hair cutting* and *office hours*, we used to have over "chaai" (sponsored by Shiri) in our office. I have learned a lot from these debates and from them and I thank them for that. Shiri was always there in moments I needed him the most. Thanks are also due to Despina Papoulis and Shiri for working with me on the implementation of SchemaLog. Some of the other friends who worked with me on various SchemaLog related projects include Mazda Boniface, Shawn Delaney, Balazs Kegl, Olga Kirianova, Krishna Kumar, Girish Mahanta, Priya Parimelalagar, Pawel Rodziewicz, and Ahmed Setar Uddin. I would like to thank them for their active participation in these projects.

In the last year of my stay at Concordia, I particularly enjoyed interacting with the new members of the database group, Fred Gingras, Ioana Ursu, and Andrea Gantchev. I thank them for the many wonderful moments we shared together and for their assistance in some of the projects I worked before leaving Concordia. I hope, I will have a chance to work with them again!

I thank the members of the analysts pool, in particular, Michael Assels, Paul Gill, and Stan Swiercz for their efficient systems support. I also thank the secretaries Edwina Bowen, Halina Monkiewicz, and Stephanie Roberts for all their help and for making the department a fun place to be in.

When I arrived in Montreal in the cold New Year eve of 1992, Ramesh Achuthan warmly hosted me and made my life easy for the first few months of my stay here. Since then, he and his wife Jaya, another one of my great friends in Montreal, have helped me in many ways and I am grateful to them.

I also thank, Gokul, Krishnan, Rama, Ravi, Sanjay, Sanjoo, Sridhar, Srini, Subram, Thankaraj, and Venki for putting up with me (and my cooking during the paper deadline periods) as a room-mate at various stages of my stay in Montreal. Sridhar deserves special mention for (the incredible feat of?) being my room-mate for two years. The arguments I had with him have always been fun and made me a bit wiser in the end. I also thank my other friends in Concordia, Behram, Birdie, Jababo, Kandy, Karima, Karun, Nick, Prasad, Rajini, and Subha, who in their own ways have taught me various things in life.

Peter MacAskil has been my spiritual guru throughout my stay in Concordia. The Thursday noon meetings I used to enjoy with him have always been a source of encouragement. I thank him and his wife Joyce for their love and affection.

My friends from Birla Institute of Technology (my alma mater), Ajay, Amit, Arijit, and Rusty have been in touch with me almost on a daily basis, often contributing to my perennial disk quota problem with the stories they send, pulled off the net. I thank them for many amusing stories and for reminding me that life is not just the one in front of the computer.

Towards the end of my thesis, I had the fortune of meeting many new friends in Silicon Valley - Thanks Ashish, Bala, and Nita for encouraging me to "finish the damn degree fast" and get over to the Bay Area!

I would like to gratefully acknowledge the financial support I received from the Concordia Graduate Fellowship, International Student Fee Remission, and the NSERC and FCAR grants obtained through Laks and Fereidoon. I also have made use of many of these awards for attending conferences. I thank Concordia and these agencies for the support.

Finally, I thank my parents S. Narayana Iyer and P. Lalithambika for instilling in me the natures of curiosity and inquisitiveness that have brought me this far. I also thank my aunt Vijaya and uncle Venkataraman for nurturing my academic ambitions since my formative years and for being a great source support in all my endeavors. My brothers Ganesh and Girish, and sister Radhika have been a constant source of love, encouragement, and support. My loving family has tolerated me for being away and has forgiven me for not being close to them whenever they needed me the most. This thesis would not have been possible without their love, affection, prayers, and the faith they had in me.

Contents

List of Figures

Chapter 1

Introduction

Tremendous advances in recent times in the areas of storage, retrieval, and processing of electronic information, in conjunction with similar advances in the computer and telecommunication industries, has brought about what the pundits refer to as the *information revolution*. As it is marching towards the third millennium, expanding in proportions hitherto unheard of, the information technology (IT) industry is having its impact felt on all facets of human life in almost every part of the world. Crucial to the continued growth of the IT industry is the development of frameworks, tools, and techniques to integrate information from multiple *kinds* of data sources spread over a distributed network and encompassing a heterogeneous mix of computers, operating systems, communication links, and local information systems. This thesis is set in this context.

Electronic information resides in various mediums, the most common of which is *database systems*. Database systems include legacy systems such as network and hierarchical databases as well as traditional systems such as relational and more recently object-oriented databases. Typically, users store, retrieve, and manipulate the data in these sources via application programs (as in the case of a network database that maintains details such as business type, CEO, and location on the Fortune 500 companies), or using a common standardized query language (such as SQL as in the case of a relational database that stores monthly operational details of various companies trading in stock exchanges in North America).

A colossal amount of information also resides in non-traditional sources such as semi-structured documents, spreadsheets, and importantly, the information sources on the World Wide Web. Users interact with these sources via user interface facilities specific to each of these sources (such as the form-based interface of a web server that provides up to date stock prices of various companies, in exchanges all over the world).

Increasingly. there is a compelling need for *interoperation* among these various information sources. An example is that of an investment consultant who would like to access and analyze information from the Fortune 500 network database. the monthly operational details relational database. and the stock price Web server, in order to take judicious decisions involving stock investments. Importantly, the consultant should be able to access the information that (s)he needs, in a uniform and seamless manner. without having to worry about the idiosyncrasies (such as geographical location, local query languages. user interfaces etc.) of each of the information sources.

This thesis addresses some important issues that arise in the manipulation of (electronic) information stored in diverse information management systems. In particular. it is motivated by the following limitations of the existing frameworks for information management.

1.1 Limitations of Existing Frameworks for Information Management

We identify three important limitations of currently available information management systems.

- The rapid progress in the field of information technology has resulted in the evolution of diverse information storage environments with data and application programs generated specifically to each of these environments. but typically incompatible with one another. This has resulted in an *inability to share data and programs across the different platforms*, the need for which has become compelling. Thus, there is a need for interoperable systems, capable of pooling together and reasoning over data from multiple kinds of information systems. Currently available technology is incapable of addressing this need.

- While seamless interoperability among multiple information sources is an indispensable need and a major challenge. problems exist even in the context of a *single* information source. In the currently available frameworks for information management (including the most popular and commercially successful relational model), the notion of what constitutes information, in the sense of what can be queried or manipulated, is restricted to what is perceived as *data* in the framework: existing technology *does not allow for manipulating the meta-data (schema) component of an information source, in the same vein as data can be manipulated.* This can be seen to be a major shortcoming, given that meta-data constitutes an important part of the information in an information source. A closely related issue is that, the expressive power of most query languages including the relational query languages, is dependent on the way information is organized in an information source - queries expressible against one schema can no longer be expressed if the *same* info is structured according to a different schema. This is clearly an undesirable aspect of existing frameworks for information management.

- Structured information sources (such as relational databases) form only a fraction of the electronic sources of information. A vast amount of information in the real world is present in a semi-structured fashion, with no clearcut notion of a schema associated with the information. A powerful example of this is the World Wide Web, which is a collection of a colossal amount of both structured and semi-structured information. Currently, there is *little support for querying and manipulating such semi-structured information sources.* Harnessing the potential of this rich source of information calls for models and languages for semi-structured information sources.

We now amplify each of the above issues with the aid of real-life examples. In the following, by a *federation*, we mean a collection of information sources requiring interoperation among themselves. In this example, we consider a federation consisting of relational databases. This federation will be used as the running example in Chapters 1 - 4 of this thesis.

Example 1.1.1 *Consider a federation of university databases consisting of relational databases univ_A, univ_B, and univ_C corresponding to universities A, B, and C. Each database maintains information on the university's departments, staff, and their average salary. Figure 1 shows a sample instance of this federation.*

univ-A

pay-Info

category	dept	avg-sal
Prof	cs	70,000
Assoc Prof	cs	60,000
Secretary	cs	35,000
Prof	Math	65,000
.	.	.

univ-B

pay-Info

category	cs	Math
Prof	80,000	65,000
Assist Prof	45,000	42,000
Asso Prof	65,000	55,000
.	.	.

univ-C

cs

category	avg-sal
Prof	65,000
Assist Prof	40,000
.	.

ece

category	avg-sal
Secretary	30,000
Prof	70,000
.	.

Figure 1: University Databases

The *univ_A* database has a single relation *pay_info* which has one tuple for each department and each category in that department. The database *univ_B* also has a single relation (also *pay_info*), but in this case, department names appear as attribute names and the values corresponding to them are the average salaries. *univ_C* has as many relations as there are departments, and has tuples corresponding to each category and its average salary in each of the *$dept_i$* relations.

The heterogeneity in these representations is evident from the example[1]: The atomic values of *univ_A* (*$dept_i$s*) appear as attribute names in *univ_B* and as relation names in *univ_C*.

It is natural for the users of one of these databases to interact with the other databases in the context of the federation of universities. The user may wish to express queries such as the following.

(Q_1) *"Which are the departments that have an average salary above \$45K in all the three universities for any given category?"*

(Q_2) *"List departments with the same name in univ_B and univ_C that have the same average salary for the same categories of staff."*

[1]We are taking a simplified version of the problem by assuming the 'names' to be the same across the databases. In reality this might not be so; *e.g.* *$dept_i$* in one database/relation might correspond to *$department_i$* in another. But this issue can be suppressed here without loss of generality as such "name mappings" can be easily realized in our framework.

Query Q_1 requires an access to each of the three databases in the federation, and manipulation of information across these databases. Note that the 'information' being manipulated does not come from the data position alone. In order to process Q_1, we also need to manipulate the schema components - attribute names in univ_B and relation names in univ_C - that contain the department names. Also, note that expressing this query in SQL would involve posing (a series of) individual SQL queries to each of the component databases and finally applying another (series of) SQL queries on the intermediate results. This is clearly undesirable and the federation user should be able to express this query in a concise and uniform way, preferably in one shot.

As illustrated by the above example, a basic need that arises in the context of a federation of information systems is *interoperability*. Interoperability can be defined as the ability to uniformly *share*, *interpret*, and *manipulate* information across component information systems in a federation. Almost all aspects of heterogeneity in a federation (*e.g.*, database schemas, data models, communication protocols, query processing, consistency management, security management, etc.) raise challenges for interoperability. Interacting with component information systems in a federation calls for the ability to query them in a manner independent of the discrepancies in their structure and data semantics. In this thesis, one of the important issues we focus on is: *how to query component information systems of a federation which store semantically similar data using dissimilar schema?*

Now, let us turn our attention to a single database scenario and consider a simple query that lists "the names of all the departments that have an average salary greater than \$50K for any category of staff". Against univ_A, this query can be expressed in a straightforward manner in SQL. However, if the information in univ_A is structured according to the schema of univ_B (the department names that are part of the answer, now occupy the attribute position), it is not straightforward to see how this query can be expressed in SQL. Note that in making this claim we are assuming that the system tables are not accessible to the end user, an assumption that did not restrict SQL from expressing the query against univ_A. Thus, it is evident that the expressive power of SQL is dependent on the way information is structured in a database, and if the information is reorganized, queries that could be posed against the original database can no longer be expressed against the restructured one. Clearly, *besides having the ability to manipulate schema as well as data, the language we develop should not* *suffer from the drawback that its expressive power is dependent on the schema of the information source.*

Finally, assume that one of the components of the university federation, univ_B, is not a relational database, but an information source on the Web which is a HTML (Hypertext Markup Language) document containing the pay information. Note that the HTML documents 'structure' information by means of tags embedded in the document. The federation user might still want to express queries of the nature of Q_1 and Q_2. Doing so in such a complex scenario requires the ability to express queries over both structured and semi-structured information sources (such as the HTML documents, flat files, etc). Thus, *our framework should be capable of handling both structured as well as semi-structured information sources in a uniform manner.*

This thesis is an outcome of our efforts in addressing these shortcomings of existing technology. It critically analyzes these issues, studies the foundations of meta-data management, interoperability, and querying and restructuring of structured as well as semi-structured information systems, and proposes models, languages, and algorithms that handle these issues in a seamless way.

The rest of this chapter is organized as follows. In Section 1.2, we undertake an extensive survey of interoperability related literature. In Section 1.3, we highlight the major contributions of this thesis and indicate how the thesis is organized. Finally, we conclude this chapter in Section 1.4, by listing some of the salient publications resulting from this thesis.

1.2 Related Works on Integration of Information

Previous work in the area of integrating information from multiple information sources can be classified into *(a)* research on multidatabase systems and *(b)* research on handling semi-structured information sources.

1.2.1 Multidatabase Systems

The objective of a multidatabase system (MDBS) is to provide end users with an integrated view of the data in multiple *database* systems. Multidatabase systems, also referred to as Heterogeneous Database Systems (HDBS) and Federated Database Systems (FDBS) by different authors, has been an active area of research for many years (see [LR82, DH84, BLN86, DP90] for some early works). Surveys in this field can be found in [ACM90] (in particular, see Sheth and Larson [SL90], Litwin, Mark, and Roussopoulos [LMR90]), Hsiao [Hsi92], and [HBP94].

The approaches attempted so far for interoperability in MDBS are based on one of the following: (i) a *common data model*, and (ii) *non-procedural languages*. In the following, we survey representative works in each of these approaches. For a comprehensive survey of related literature, the reader is referred to [LMR90] and [ACM94].

Common data model: The databases participating in the federation are mapped to a common data model (CDM) (such as the object-oriented model, that naturally meets the CDM requirements in terms of richness of modeling power) which acts as an 'interpreter' among them. The similarities in the information contents of the individual databases and their semantical inter-relationships are captured in the mappings to the CDM. In such a setting, the user queries the CDM using a CDM language, and usually has to be aware of the CDM schema. In a more sophisticated scenario, 'views' which correspond to the schema of the participating databases are defined on the CDM, thus providing the user with a convenient illusion that all the information she gets is from her own database. (This is called *tight coupling* [SL90].)

A "canonical" example of the CDM-based approach is the Pegasus project of Ahmed *et. al.* [ASD+91]. Pegasus defines a common object model for unifying the data models of the underlying systems. Landers and Rosenberg use the functional model of DAPLEX as the CDM in their *Multibase* project [LR82]. *Mermaid* (Templeton *et. al* [Tem87]) uses a relational CDM, and allows only for relational interoperability (with extensions to include text). Thus users of a Mermaid federation may formulate queries using SQL.

The major problem associated with the approaches in this category is the amount of human participation required for obtaining the CDM mappings. Dynamic changes in semantics or the schemas of the individual databases can also lead to revisions in the CDM (mappings) requiring major (and hence costly) human intervention. Also in many cases, autonomy requirements might impose limits on the information available for constructing the CDM mappings.

Non-procedural languages: The second approach for interoperability in MDBS involves defining a language that can allow users to define and manipulate a collection of autonomous databases in a non-procedural way. Thus a CDM, as defined in the previous case is not required; the non-procedural language in some sense plays the role of the CDM here. The major advantage associated with this approach is the flexibility such a *loose coupling* ([SL90]) provides.

Litwin, Mark, and Roussopoulos [LMR90], advocate that *the concept of an MDBS language is central to the notion of a MDBS system.* They argue against a global interpretation (as obtained in the CDM approach), and discuss the merits of a language MSQL (Multidatabase-SQL) [Lit89, GLRS93], an extension of SQL for interoperability among multiple relational databases. The salient features of this language include the ability to retrieve and update relations in different databases, define multidatabase views, and specify compatible and equivalent domains across different databases. In ([CL93]) Chomicki and Litwin propose an extension to OSQL ([ADK+91]), a functional object-oriented language. The language has constructs that are capable of declaratively specifying a broad class of mappings across multiple object-oriented databases. They also sketch the operational semantics of this language.

More recently, Sciore, Siegel, and Rosenthal [SSR94], introduce a theory of *semantic values* as a unit of exchange that facilitates interoperability. They apply this theory on the relational model, and propose an extension to SQL called context-SQL (CSQL). For each attribute in a schema, CSQL provides the capabilities for specifying, and manipulating *meta-attributes*, that correspond to properties of that attribute. These meta-attributes provide the context information for interoperability among databases in an MDBS. UniSQL/M [KGK+95] is a multidatabase system for managing a heterogeneous collection of relational database systems. The language of UniSQL/M, known as SQL/M, provides facilities for defining a global schema over related entities in different local databases, and to deal with simple semantic heterogeneity issues such as scaling and unit transformation. The emerging standard for SQL3 ([SQL96, Bee93]) supports abstract data typess and object id's, and hence shares some of the features of the previously discusses languages.

Languages based on higher-order logic have been used as a vehicle for interoperability. The underlying philosophy is that the schematic information should be seriously considered as part of a database's information content. Thus such approaches are especially suited for handling *schematic discrepancies* ([KLK91]) commonly occurring in MDBS. These approaches involve defining a higher-order logical language that can express queries ranging over the (meta)information corresponding to the individual databases and their schemas. The major advantage associated with such approaches is the declarativity they derive from their logical foundations.

Lefebvre, Bernus, and Topor [LBT92] use F-logic (Kifer *et. al.* [KLW95]), to provide data mappings between local databases and an *assumed* global database (that is an integrated view of the local databases). The mappings take care of the data as well as the schema discrepancies in the local databases. Global queries are translated into local queries via a query translation algorithm, also written in F-logic. The major strength of this approach is that using a declarative medium to provide the mapping as well as the query translation rules helps in conciseness, modularity, and maintenance. Krishnamurthy and Naqvi [KN88] propose a Horn-clause like language that can "range over" both data and meta-data by allowing "higher-order" variables. Krishnamurthy, Litwin, and Kent [KLK91] extend this language and demonstrate its capability for interoperability. However, they do not provide a formal model-theoretic or proof-theoretic semantics for their language, and their language is not a full-fledged logic.

An approach that falls in between the above two classifications is the M(DM) model of Barsalou and Gangopadhyay [BG92]. M(DM) deals with a set of metatypes that formalize data model constructs in second-order logic. A data model is hence a collection of M(DM) metatypes. A schema instantiates these metatypes into a set of first-order types. A database then consists of instances of the schema types. M(DM)'s metatypes are organized into an inheritance lattice[2] which provides extensibility.

One of the earliest proposals for treating data and meta-data in a uniform manner, comes from Codd, the inventor of relational model. In [Cod79], Codd extends the relational model and algebra in such a manner that the model captures more meaning than captured by the basic relational model. A notable operator in this extended-relational algebra is an operator that takes a relational extension as its argument and returns the corresponding relation name. By allowing for the relation names in the database to be queried (besides allowing for querying the values), the algebra of [Cod79] blurs the distinction between data and meta-data.

Approaches discussed so far suffer from one or more of the following drawbacks. In order to effectively handle schematic discrepancies, the schema information should be given primary status (along with values appearing in the databases) within the

[2]The term "lattice" used here is not in its mathematical sense; it is loosely used by the authors to put forth their ideas.

language. This functionality is lacking in some of the above approaches ([Lit89, SSR94, KGK$^+$95]). In these approaches, as there is no uniform treatment of data and meta-data, schema browsing and specifying "higher-order mappings" would be inconvenient. While [LBT92] makes use of the higher-order capabilities of F-logic in uniformly manipulating data and schema, their approach uses F-logic primarily for query translation and provides an SQL-based user interface. This severely limits the schema browsing capabilities from the user interface. Approaches that base interoperability on higher-order mappings among component databases ([CL93, LBT92]) do not provide support for ad-hoc queries that refer to (and possibly compare) data and schema components of multiple databases in one shot. [CL93] and [SSR94] do not provide a uniform syntax for multiple databases in a manner that glosses over their schematic discrepancies. The SQL3 standard ([SQL96, Bee93]), while a computationally complete language, to our knowledge, does not have facilities for *directly* supporting the kind of higher-order mappings required in an MDBS setting. While [Cod79] attempts to provide equal status to the values and the relation names in a database, it does not extend the same treatment to other schema components such as the attribute names and the database name. In [BG92], although the combination of logic, object orientation, and metaprogramming gives much power to the M(DM) model, its second-order nature raises questions about the possibility of practical implementations based on this approach. Also, its semantics is quite complex.

1.2.2 Semi-structured Information Sources

Interoperability among semi-structured information systems is an active area of current research. Components of an MDBS are well-structured data sources with a characteristic support for standard query languages. However, semi-structured information sources are typically exposed to the outside world via interfaces unique to each of the sources and hence incompatible with one another. Some of these sources need not even be information systems, in the traditional sense (*e.g.* spreadsheets, mail tools). Thus, realizing interoperability among semi-structured data sources poses a considerable challenge. Wiederhold [Wie92] proposed the important concept of a *mediator* - a program that integrates heterogeneous data sources, and provides a uniform interface through which end users may ask queries. A mediator queries each source using an interface, called a *wrapper*, provided by that source. Informally, a wrapper provides a *view* of the associated information source, and implements a class of queries against it. The mediator then integrates the data coming from disparate sources in a uniform manner. Several data integration projects [ACHK94, CGH+94, SAB+95, Ham94, TRV96] have developed mediator languages that an application developer can use to create a mediator. The TSIMMIS ([CGH+94]) project at Stanford is based on a 'light weight' object model called Object Exchange Model (OEM). It provides tools for generating the components (including wrappers) that are needed to build systems for integrating information. The HERMES ([SAB+95]) project at Maryland is broader in scope and considers, besides databases, arbitrary software packages, multimedia sources etc. IBM's Garlic project ([CHS+95]) also provides substantial support for multimedia reasoning and integration.

While the TSIMMIS approach caters to a variety of semi-structured sources whose scheme may be ill-defined or even unknown, it is a framework for the mediator *developer* and is too low level to serve the needs of an end user. Also, it intertwines the information access and integration activities; the integration strategy dynamically varies depending on the data being fetched. Thus, the TSIMMIS goal is not to perform a fully automated integration, but rather to provide a framework and tools to assist developers in their integration efforts.

The HERMES project aims to cater to a wide variety of data sources and makes use of 'ports' exposed by the component data sources, for developing mediator applications. Thus, the power of this approach is limited by the capabilities of the data sources. For example, the HERMES approach to interoperability among multiple relational databases, would make use of the SQL ports provided by the relational sources to despatch SQL queries against the sources and integrate the returned results. Thus, the interoperability application would be constrained by the limitations of SQL.

Recently, Gyssens at al. [GLS95, GLS96] proposed a two-dimensional data model called the tabular data model and defined a tabular algebra corresponding to it. They also show that their algebra is complete for querying and restructuring. They observed the potential of the tabular model and algebra to serve as a unifying model for relations (as in databases) and spreadsheets. However, this proposal is tailored for tabular information, and does not consider non-tabular information sources requiring interoperability.

In [LSK95], Levy, Srivastava, and Kirk present an architecture for query processing in global information systems. Their approach is based on description logic. While their framework is more general than that of traditional MDBS, many of the issues they study also arise in MDBS and their techniques are applicable to MDBS. From this perspective, their approach is based on mapping the "component" information systems to a so-called "world-view", which is similar to a CDM. Query optimization being their main concern, issues such as schema browsing, restructuring, and database programming are not addressed.

In recent times, the problem of querying and updating semi-structured data stored in documents has received much attention ([ACM93, CAS94, ACM95, Abi97]). These works are relevant in our context for the reason that many common data sources requiring interoperability such as the Web, mail folders etc, are in fact a collection of semi-structured documents. In [ACM93], Abiteboul, Cluet, and Milo make use of the grammar of a document to 'map' it to an appropriate object-oriented database. Christophides, Abiteboul, Cluet, and Scholl ([CAS94]) use a similar idea to map SGML (Standard General Markup Language) documents into object-oriented databases. However, these proposals do not account for novel features such as navigation (via hyperlinks), essential for querying information sources on the Web.

Many recent proposals ([LRO96, MMM96, AMMT96]) have addressed the challenges arising in the context of Web querying. Most of these proposals are specific to the Web setting and it is unclear how they can be deployed in a scenario requiring interoperation among Web as well as traditional data sources. In Chapter 5, we revisit this issue and study these works in detail.

1.2.3 Our Approach

Our approach in this thesis is based on using (higher-order) languages as a vehicle for integrating information from multiple data sources. In particular, we develop these languages with an aim towards addressing the drawbacks of existing frameworks for information management, identified in Section 1.1.

We believe that *declarativity* is a key requirement for interoperability among component data sources in a federation. A logic-based approach for interoperability would bring the advantages of clear foundations, sound formalism, and proof procedures thus providing for a truly declarative environment. Conventional query languages are based on predicate calculus and are useful for querying the *data* in a data source. But as discussed in Section 1.1, interoperability necessitates a functionality to query not only the data in a source but also its schema or meta-data. This calls for a higher-order language which treats "components" of such meta-data as "first class" entities in its semantic structure. In such a framework, queries that manipulate data as well as their schema "in the same breath" could be naturally formulated. However, some of the important concerns in designing an expressive logic language are the following. The language should (1) be sound and complete; (2) be tractable in admitting simple and efficient proof procedures and an effective implementation; (3) enhance the expressive power significantly while adding relatively few simple constructs to first-order logic; and (4) admit queries and programs to be expressed intuitively and concisely. Some of the other key features required in a language for interoperability in a federation of heterogeneous information sources are the following. The language must (5) have an expressive power that is *independent* of the *schema* with which the source is structured; (6) to promote interoperability, permit the *restructuring* of one source to conform to the schema of another; (7) be easy to use and yet sufficiently expressive; (8) be capable of handling both structured and semi-structured information sources in a uniform manner; and finally (9) admit effective and efficient implementation. In particular, it must be possible to realize a *non-intrusive* implementation that would require *minimal additions* to component information management systems. In the rest of this thesis, we show how these goals can be realized.

1.3 Structure and Contributions of This Thesis

I. In Chapter 2 of this thesis, we develop a formal framework for interoperability based on a higher-order logic called SchemaLog. Though this thesis is set in the general context of structured as well as semi-structured information sources, we begin our study by developing SchemaLog as a foundation for interoperability among *relational* databases. We have chosen the relational model for this purpose for *(a)* it has sound theoretical foundations, *(b)* it is perhaps the best understood data model, and *(c)* it is commercially popular and widely available. Later in the thesis, we extend our study to other (more general) settings involving non-relational and semi-structured information sources.

Our other contributions in this chapter include the development of a fixpoint theory for the definite clause fragment of SchemaLog and demonstration of its equivalence to the model-theoretic semantics. We also develop a sound and complete proof procedure for *all* clausal theories. We establish the correspondence between SchemaLog and first-order predicate calculus and provide a reduction of SchemaLog to predicate calculus. We illustrate the simplicity and power of SchemaLog with a variety of applications involving database programming (with schema browsing). schema integration. schema evolution. cooperative query answering, and aggregation. We also highlight our implementation of SchemaLog realized on a federation of INGRES databases.

II. Crucial to realizing an efficient implementation of SchemaLog is the study of the fundamental operations underlying the logical language. Thereto. in Chapter 3. we develop an algebra capable of querying and restructuring relational databases. We also develop a calculus language inspired by a fragment of SchemaLog useful for federation querying, and develop a notion of varying levels of safety that naturally arise in the context of our calculus. Our results in this chapter, on the equivalence between the various safe fragments of the calculus language and fragments of the algebra, form the theoretical foundations of an effective implementation of SchemaLog.

III. SQL is the lingua franca of the database community. especially for the practitioners in the industry. While SchemaLog is a powerful language rooted on sound theoretical foundations. many of its (mostly syntactic) features are hard to adapt to for a typical. SQL "die-hard" database programmer. However. it would be easy for such a user to adapt to a language having a syntax closer to SQL. but which retains the spirit of SchemaLog. Dictated by these practical considerations. in Chapter 4, we develop an SQL-like language called *SchemaSQL* that is capable of querying as well as restructuring, both data and schema[3]. *SchemaSQL* is also capable of performing novel aggregation (beyond the realm of what is possible in SQL). along the lines of the complex aggregation operations commonly performed in On-line Analytical Processing (OLAP) systems. We also propose an implementation architecture for *SchemaSQL* that is

[3]ANSI SQL92 is the dialect of SQL we consider as our starting point.

designed to build on existing technology and present an algorithm for realizing a system based on *SchemaSQL* .

IV. In Chapter 5, we demonstrate how the thoughts that went into the design of SchemaLog can be brought to bear in a natural way, in the context of legacy databases (such as the Network databases) and non-traditional information systems (such as the World Wide Web). We show that the powerful features of SchemaLog for meta-data querying, information restructuring, and interoperability can be naturally extended to a truly heterogeneous federation consisting of both relational and non-relational sources. Thus, this chapter demonstrates the role of SchemaLog as a foundation for integrating heterogeneous information sources.

Finally, we discuss future research and conclude in Chapter 6.

A Note On The Background Assumed Of The Reader: While the prerequisites for an understanding of the thesis have been kept to a minimum, Chapter 2 assumes a knowledge of first-order logic (See Chang and Lee [CL73], Enderton [End72]). In particular, many of the proofs presented in this chapter, follow the proof techniques of [CL73]. Chapter 3 assumes a basic knowledge of Relational Algebra and Relational Calculus, and Chapter 4 assumes some familiarity with SQL. Finally, an understanding of the ER-model and the Network model is assumed for Chapter 5.

Before closing this chapter, we list selected publications that have resulted from this thesis.

1.4 Selected Thesis Related Publications

1. Laks V.S. Lakshmanan, F. Sadri, and Iyer N. Subramanian. "Logic and Algebraic Languages for Interoperability in Multidatabase Systems", Accepted to the *Journal of Logic Programming*, February 1996.

2. Marc Gyssens, Laks V.S. Lakshmanan, and Iyer N. Subramanian. "The Tabular Data Model", Submitted to *ACM Transactions on Database Systems*, November 1996.

3. Frédéric Gingras, Laks V.S. Lakshmanan, Iyer N. Subramanian, Despina Papoulis, and Nematollaah Shiri. "Languages for Multidatabase Interoperability", In *Proc. of the ACM SIGMOD Conf.* Tucson, AZ. May 1997 (Tools Demo).

4. Alanoly Andrews, Laks V.S. Lakshmanan, Nematollaah Shiri, and Iyer N. Subramanian. "On Implementing SchemaLog - An Advanced Database Programming Language", In *Proc. of the Fifth International Conference on Information and Knowledge Management* (CIKM'96), Rockville, Maryland, Nov 1996.

5. Laks V.S. Lakshmanan, F. Sadri, and Iyer N. Subramanian. "SchemaSQL - A Language for Querying and Restructuring Multidatabase Systems". In *Proc. of the International Conference on Very Large Databases* (VLDB'96), Bombay, India, Sept 1996.

6. Marc Gyssens, Laks V.S. Lakshmanan, and Iyer N. Subramanian. "Tables as a Paradigm for Restructuring Information", In *Proc. of the 15th ACM Symposium in Principles of Database Systems* (PODS'96), Montreal, Canada, June 1996.

7. Laks V.S. Lakshmanan, F. Sadri, and Iyer N. Subramanian. "A Declarative Language for Querying and Restructuring the Web", In *Proc. of the 6th International Workshop on Research Issues on Data Engineering* (RIDE-NDS) at ICDE, New Orleans, Louisiana, February 1996.

8. Laks V.S. Lakshmanan, Iyer N. Subramanian, Despina Papoulis, and Nematollaah Shiri. "A Declarative System for Multidatabase Interoperability", In *Proc. of the 4th International Conference on Algebraic Methodology and Software Technology (AMAST)*, Montreal, Canada, July 1995 (Tools Demo).

9. Laks V.S. Lakshmanan, F. Sadri, and Iyer N. Subramanian. "On the Logical Foundations of Schema Integration and Evolution in Heterogeneous Database Systems", In *Proceedings of the Third International Conference on Deductive and Object Oriented Database Systems (DOOD)*, Phoenix, Az, December 1993.

Technical Reports

10. Laks V. S. Lakshmanan, Iyer N. Subramanian, Nita Goyal, and Ravi Krishnamurthy. "On Querying and Updating the Spreadsheets", *Technical Report*, Dept of Computer Science, Concordia University, October 1996.

11. Laks V. S. Lakshmanan, Iyer N. Subramanian, and V.S.Subrahmanian. "Conflict Management in Mediators for Heterogeneous Data Sources", *Technical Report*, Dept of Computer Science, Concordia University, October 1996.

12. Laks V. S. Lakshmanan, Despina Papoulis, and Iyer N. Subramanian. "Realizing Schemalog", *Technical Report*, Dept of Computer Science, Concordia University, August 1994.

Chapter 2

SchemaLog

In this chapter, we develop the logical foundations of interoperability among component data sources in a federation, based on a higher-order logic called *SchemaLog*. SchemaLog provides uniform status to data as well as schema components of the sources. Thus, it allows for naturally formulating queries that manipulate data as well as their schema in a uniform manner. SchemaLog, besides being a sound and complete logic, is tractable in admitting simple and efficient proof procedures and an effective implementation, enhances the expressive power significantly while adding relatively few simple constructs to first-order logic, and admits queries and programs to be expressed intuitively and concisely.

2.1 Overview of the Chapter

- We introduce SchemaLog informally with a motivating example (Section 2.2). Our syntax (Section 2.3.1) was inspired in part by that of [KN88]. However while they provide no formal semantics, we develop model-theoretic (Section 2.3.2), fixpoint (Section 2.5.1), and proof-theoretic (Section 2.5.2) semantics for SchemaLog. Thus, unlike their language, SchemaLog is a full-fledged logic. Besides, technically the framework developed by us is different from that of [KN88].

- We propose a proof procedure for full clausal SchemaLog and show that it is sound and complete (Section 2.5.2).

- SchemaLog, like *HiLog* (Chen *et. al.* [CKW90]), is syntactically higher-order but semantically first-order. We give a reduction of SchemaLog to first-order predicate calculus (Section 2.4). This reduction yields the technical benefits of soundness, completeness, and compactness for SchemaLog. However we argue that for interoperability, a crucial requirement for a query language is "*schema preservation*" (see Section 2.4), and prove that under this requirement SchemaLog has a strictly higher expressive power than first-order logic.

- We illustrate a number of applications of SchemaLog for practical problems. in the field of MDBS as well as schema browsing, cooperative query answering, schema evolution and integration, and computation of powerful forms of aggregation beyond the abilities of conventional languages like SQL (Section 2.6). We also outline the potential of SchemaLog for providing a theoretical foundation for OLAP (On-Line Analytical Processing), currently an active area of research with tremendous practical potential.

- We compare SchemaLog with previously proposed higher-order logics including F-logic and HiLog and bring out its unique features (Section 2.7).

- We briefly highlight our implementation of a SchemaLog-based interoperable system on a federation of INGRES databases (Section 2.8). Finally, we summarize the chapter and discuss future research (Section 2.9).

In this chapter, we confine ourselves to the interoperability problem in relational databases. Our eventual objective (achieved in Chapter 5) is to extend SchemaLog into a logic capable of providing for interoperability among different data models. In the rest of the chapter, by a *federation*, we mean a collection of relational databases that can interoperate among themselves.

2.2 SchemaLog By Example

In this section, we introduce the syntax and intuitive semantics of our proposed language informally via the university federation example (Example 1.1.1) of Chapter 1. We will follow it with a formal account of the syntax and semantics in the next section.

Let us revisit the queries Q_1 and Q_2 of Example 1.1.1.

(Q_1) *"Which are the departments that have an average salary above \$45K in all the three universities for any given category?"*

(Q_2) *"List departments with the same name in univ_B and univ_C that have the same average salary for the same categories of staff."*

Each database is made of relations, and each relation is made of tuples, which are functions mapping attributes to values. Identification of the set of tuples which constitute a relation could be accomplished by associating tuple-id's with them. Now the query Q_1 can be expressed in SchemaLog as[1]:

$? - univ\_A :: pay\_info[T_1 : dept \rightarrow D,\ category \rightarrow C,\ avg\_sal \rightarrow S_1],$
$univ\_B :: pay\_info[T_2 : category \rightarrow C,\ D \rightarrow S_2],\ D \neq \text{`}category\text{'},$
$univ\_C :: D[T_3 : category \rightarrow C,\ avg\_sal \rightarrow S_3],\ S_1 > 45K,\ S_2 > 45K,\ S_3 > 45K$

---

and query Q_2 can be expressed as:

$? - univ\_B :: pay\_info[T_1 : category \rightarrow C,\ D \rightarrow S],\ D \neq \text{`}category\text{'},$
$univ\_C :: D[T_2 : category \rightarrow C,\ avg\_sal \rightarrow S]$

Notice that in query Q_1, variable D ranges over domain values as well as attribute and relation names. It is this flexibility which makes such a querying medium highly expressive and declarative. The variables T_i intuitively stand for the tuple-id's corresponding to the tuples in the relations.

In queries Q_1 and Q_2, the variable D is explicitly compared with the attribute *category* whenever D occurs in a position which ranges over attributes. Thus an explicit comparison is required, unless it is known, *e.g.* that there is no relation called *category* in *univ_C*.

2.3 SchemaLog – Syntax and Semantics

In this section we formally present the syntax and semantics of our language.

[1]Existential variables can be projected out by writing rules. Here, we mainly focus on the intuition behind the syntax of SchemaLog.

2.3.1 Syntax

We use strings starting with a lower case letter for constants and those starting with an upper case letter from the end of the alphabet (such as $X, Y, \ldots$) for variables. As a special case, we use t_i to denote arbitrary terms of the language. $\mathcal{A}, \mathcal{B}, \ldots$ denote arbitrary well-formed formulas and $A, B, \ldots$ denote arbitrary atoms.

The vocabulary of the language $\mathcal{L}$ of SchemaLog consists of pairwise disjoint countable sets $\mathcal{G}$ (of function symbols), $\mathcal{S}$ (of non-function symbols), $\mathcal{V}$ (of variables), and the usual logical connectives $\neg, \vee, \wedge$, and quantifiers $\exists$ and $\forall$.

Every symbol in $\mathcal{S} \cup \mathcal{V}$ is a term of the language. If $f \in \mathcal{G}$ is a n-place function symbol, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

An *atomic formula* of $\mathcal{L}$ is an expression of one of the following forms:

$\langle\text{db}\rangle::\langle\text{rel}\rangle[\langle\text{tid}\rangle : \langle\text{attr}\rangle \rightarrow \langle\text{val}\rangle]$

$\langle\text{db}\rangle::\langle\text{rel}\rangle[\langle\text{attr}\rangle]$

$\langle\text{db}\rangle::\langle\text{rel}\rangle$

$\langle\text{db}\rangle$

where $\langle\text{db}\rangle$, $\langle\text{rel}\rangle$, $\langle\text{attr}\rangle$, $\langle\text{tid}\rangle$, and $\langle\text{val}\rangle$ are terms of $\mathcal{L}$. Example 2.3.1 illustrates the intuitive meaning of this syntax. In an atom of the form

$$\langle\text{db}\rangle::\langle\text{rel}\rangle[\langle\text{tid}\rangle : \langle\text{attr}\rangle \rightarrow \langle\text{val}\rangle]$$

we refer to the terms $\langle\text{db}\rangle$, $\langle\text{rel}\rangle$, $\langle\text{attr}\rangle$, and $\langle\text{val}\rangle$ as the *non-id components* and $\langle\text{tid}\rangle$ as the *id component*. The id component intuitively stands for tuple-id (tid). The *depth* of an atomic formula A, denoted *depth*(A), is the number of non-id components in A. The depths of the four categories of atoms introduced above are 4,3,2, and 1 respectively. By our definition of atoms, an id-component appears only in atoms of depth 4. The well-formed formulas (wff's) of $\mathcal{L}$ are defined as usual: every atom is a wff; $\neg\mathcal{A}$, $\mathcal{A} \vee \mathcal{B}$, $\mathcal{A} \wedge \mathcal{B}$, $(\exists X)\mathcal{A}$, and $(\forall X)\mathcal{A}$ are wff's of $\mathcal{L}$ whenever $\mathcal{A}$ and $\mathcal{B}$ are wff's and X is a variable.

We also permit *molecular formulas* of the form

$\langle\text{db}\rangle::\langle\text{rel}\rangle[\langle\text{tid}\rangle : \langle\text{attr}_1\rangle \rightarrow \langle\text{val}_1\rangle, \ \ldots, \ \langle\text{attr}_n\rangle \rightarrow \langle\text{val}_n\rangle]$

as an abbreviation of the corresponding well-formed formula

$\langle\text{db}\rangle::\langle\text{rel}\rangle[\langle\text{tid}\rangle : \langle\text{attr}_1\rangle \rightarrow \langle\text{val}_1\rangle] \wedge \cdots \wedge \langle\text{db}\rangle::\langle\text{rel}\rangle[\langle\text{tid}\rangle : \langle\text{attr}_n\rangle \rightarrow \langle\text{val}_n\rangle]$.

In spirit, molecular formulas of SchemaLog are similar to the molecules in F-logic [KLW95], from which we borrow this term.

A *literal* is an atom or the negation of an atom. A *clause* is a formula of the form $\forall X_1 \cdots \forall X_m(L_1 \vee \cdots \vee L_n)$ where each L_i is a literal and $X_1, \ldots, X_m$ are all the variables occurring in $L_1 \vee \cdots \vee L_n$. A *definite clause* is a clause in which exactly one positive literal is present and it is represented as $A \leftarrow B_1, \ldots, B_n$, where A is called the *head* and $B_1, \ldots, B_n'$ is called the *body* of the definite clause. A *unit clause* is a clause of the form $A \leftarrow$, that is a definite clause with an empty body.

Example 2.3.1 *The molecule $univ\_B :: pay\_info[T : category{\rightarrow}C,\ D{\rightarrow}45K]$ in the context of the university federation asserts the fact that database $univ\_B$ has a relation pay_info which has an attribute category and an attribute that contains, for some tuple, a value $45K$.*

2.3.2 Semantics

Let U be a non-empty set of elements called *intensions* (corresponding to the terms of $\mathcal{L}$). Consider a function $\mathcal{I}$ that maps each non-function symbol to its corresponding intension in U and a function $\mathcal{I}_{fun}$ which interprets the function symbols as functions over U. The true atoms of the model are captured using a function $\mathcal{F}$ which takes as arguments the name of the database, the relation name, attribute name, and tuple-id, and maps to a corresponding individual value. Thus for a given atomic formula to be true, the function $\mathcal{F}$ corresponding to the formula (after mapping the symbols of the formula to their corresponding intensions) should be *defined* in the structure (and the values should match).

A *semantic structure* M for our language is a tuple $< U, \mathcal{I}, \mathcal{I}_{fun}, \mathcal{F} >$ where

- U is a non-empty set of intensions;
- $\mathcal{I}$: $\mathcal{S} \rightarrow U$ is a function that associates an element of U with each symbol in $\mathcal{S}$;
- $\mathcal{I}_{fun}(f) : U^n {\rightarrow} U$, where f is a function symbol of arity n in $\mathcal{G}$.
- $\mathcal{F} : U \rightsquigarrow [U \rightsquigarrow [U \rightsquigarrow [U \rightsquigarrow U]]]$, where $[A \rightsquigarrow B]$ denotes the set of all partial functions from A to B.

To illustrate the role of $\mathcal{F}$, consider the atom $d :: r$. For this atom to be true, $\mathcal{F}(\mathcal{I}(d))(\mathcal{I}(r))$ should be defined in M. Similarly, for the atom $d :: r[t : a \rightarrow v]$ to be true, $\mathcal{F}(\mathcal{I}(d))(\mathcal{I}(r))(\mathcal{I}(a))(\mathcal{I}(t))$ should be defined in M and $\mathcal{F}(\mathcal{I}(d))(\mathcal{I}(r))(\mathcal{I}(a))(\mathcal{I}(t)) = \mathcal{I}(v)$.

A *vaf* (variable assignment function) is a function $\nu : \mathcal{V} \longrightarrow U$. We extend it to the set $\mathcal{T}$ of terms as follows.

- $\nu(s) = \mathcal{I}(s)$ for every $s \in \mathcal{S}$.
- $\nu(f(t_1, \ldots, t_k)) = \mathcal{I}_{fun}(f)(\nu(t_1), \ldots, \nu(t_k))$, where f is a function symbol of arity k in $\mathcal{G}$ and t_i are terms.

Let $t_i \in \mathcal{T}$ be any term. The *satisfaction* of an atomic formula A, in a structure M under a vaf ν is defined as follows.

- Let A be of the form t_1. Then $M \models_\nu A$ iff $\mathcal{F}(\nu(t_1))$ is defined in M.
- Let A be of the form $t_1 :: t_2$. Then $M \models_\nu A$ iff $\mathcal{F}(\nu(t_1))(\nu(t_2))$ is defined in M.
- Let A be of the form $t_1 :: t_2[t_3]$. Then $M \models_\nu A$ iff $\mathcal{F}(\nu(t_1))(\nu(t_2))(\nu(t_3))$ is defined in M.
- Let A be of the form $t_1 :: t_2[t_4 : t_3 \rightarrow t_5]$. Then $M \models_\nu A$ iff $\mathcal{F}(\nu(t_1))(\nu(t_2))(\nu(t_3))(\nu(t_4))$ is defined in M, and $\mathcal{F}(\nu(t_1))(\nu(t_2))(\nu(t_3))(\nu(t_4)) = \nu(t_5)$

Satisfaction of compound formulas is defined in the usual way:

- $M \models_\nu (\mathcal{A} \vee \mathcal{B})$ iff $M \models_\nu \mathcal{A}$ or $M \models_\nu \mathcal{B}$;
- $M \models_\nu (\neg\mathcal{A})$ iff $M \not\models_\nu \mathcal{A}$;
- $M \models_\nu (\exists X)\mathcal{A}$ iff for *some* vaf μ, that may possibly differ from ν only on X, such that $M \models_\mu \mathcal{A}$;

For *closed* formulas, $M \models_\nu \mathcal{A}$ does not depend on ν and we can simply write $M \models \mathcal{A}$.

Before closing this section, we note that built-in predicates ($=, \neq, \leq$, *etc.*) can be introduced and interpreted in SchemaLog in the usual manner. We shall freely make use of built-in predicates in our examples.

2.4 Reduction of SchemaLog to Predicate Calculus

The richer syntax of SchemaLog may raise questions about its axiomatizability and hence its potential for being implemented as a viable medium for MDBS interoperability. In this section, we prove that every SchemaLog formula can be encoded in first-order predicate calculus, in an equivalence preserving manner. This will show that SchemaLog inherits many of the desirable properties of first-order logic, while offering the convenience of a higher-order syntax.

Syntax

We define a language $\mathcal{L}_{fol}$ that is derived from the SchemaLog language $\mathcal{L}$. $\mathcal{L}_{fol}$ consists of the set $\mathcal{S}$ of logical symbols, variables $\mathcal{V}$, the function symbols $\mathcal{G}$, and unique predicate symbols $call_1, call_2, call_3$, and $call_4$. $\mathcal{S}$, $\mathcal{V}$ and $\mathcal{G}$ correspond to those in $\mathcal{L}$.

Given a SchemaLog formula $\mathcal{A}$, its encoding $\mathcal{A}^*$ in predicate calculus is determined by the recursive transformation rules given below. In this discussion, $s_i \in \mathcal{S} \cup \mathcal{V}$, $f \in \mathcal{G}$, $t, t_{db}, t_{rel}, t_{attr}, t_{id}, t_{val} \in \mathcal{T}$, the set of terms, and $\mathcal{A}$, $\mathcal{B}$ are any formulas.

$$\begin{aligned}
&\text{encode } (s) && = s \\
&\text{encode } (f) && = f \\
&\text{encode } (f(t_1, \ldots, t_n)) && = \text{encode}(f)(\text{encode}(t_1), \ldots, \text{encode}(t_n)) \\
&\text{encode } (t_{db} :: t_{rel}[t_{tid} : t_{attr} \to t_{val}]) && = call_1(encode(t_{db}), encode(t_{rel}), encode(t_{tid}), \\
& && \quad encode(t_{attr}), encode(t_{val})) \\
&\text{encode } (t_{db} :: t_{rel}[t_{attr}]) && = call_3(encode(t_{db}), encode(t_{rel}), encode(t_{attr})) \\
&\text{encode } (t_{db} :: t_{rel}) && = call_2(encode(t_{db}), encode(t_{rel})) \\
&\text{encode } (t_{db}) && = call_1(encode(t_{db})) \\
&\text{encode } (\mathcal{A} \vee \mathcal{B}) && = \text{encode } (\mathcal{A}) \vee \text{encode } (\mathcal{B}) \\
&\text{encode } (\mathcal{A} \wedge \mathcal{B}) && = \text{encode } (\mathcal{A}) \wedge \text{encode } (\mathcal{B}) \\
&\text{encode } (\neg \mathcal{A}) && = \neg \text{ encode } (\mathcal{A}) \\
&\text{encode } (\mathcal{A} \to \mathcal{B}) && = \text{encode } (\mathcal{A}) \to \text{encode } (\mathcal{B}) \\
&\text{encode } ((QX)\mathcal{A}) && = (QX)\text{encode}(\mathcal{A}), \text{ where } Q \text{ is} \\
& && \quad \text{either } \exists \text{ or } \forall.
\end{aligned}$$

Semantics

Given a SchemaLog structure $M_s = < U, \mathcal{I}, \mathcal{I}_{fun}, \mathcal{F} >$, we construct a corresponding first-order structure, encode(M_s) = $M_{fol} = < U, \mathcal{I}_f, \mathcal{I}_p >$ as follows. ($\mathcal{I}_f$ interprets function symbols of $\mathcal{L}_{fol}$ as functions of appropriate arity over U. $\mathcal{I}_p$ interprets the predicate symbols of L_{fol} as relations of appropriate arity over U. Note that the logical symbols $s \in \mathcal{S}$ are function symbols of arity 0.)

$\mathcal{I}_f(s) =_{def} \mathcal{I}(s)$, for each $s \in \mathcal{S}$.

$\mathcal{I}_f(f)(u_1, \ldots, u_n) =_{def} \mathcal{I}_{fun}(f)(u_1, \ldots, u_n)$, for each $f \in \mathcal{G}$ of arity n, and $u_1, \ldots, u_n \in U$.

The $call_i$ predicates of L_{fol} are interpreted in the following way:

Let $d, r, t, a, v \in U$.

$\langle d, r, t, a, v\rangle \in \mathcal{I}_p(call_4)$ iff $\mathcal{F}(d)(r)(t)(a)$ is defined in M_s, and $\mathcal{F}(d)(r)(t)(a) = v$.

$\langle d, r, a\rangle \in \mathcal{I}_p(call_3)$ iff $\mathcal{F}(d)(r)(a)$ is defined in M_s.

$\langle d, r\rangle \in \mathcal{I}_p(call_2)$ iff $\mathcal{F}(d)(r)$ is defined in M_s.

$\langle d\rangle \in \mathcal{I}_p(call_1)$ iff $\mathcal{F}(d)$ is defined in M_s.

A variable assignment ν is a function from the variables of $\mathcal{L}_{fol}$ to the universe U. ν is extended to the set $\mathcal{T}$ of terms, analogously to the way it is done in Section 2.3.2. Then, the truth of a well-formed formula $\mathcal{A}$, with variable assignment ν in structure M_{fol} is defined as follows:

1. If $\mathcal{A}$ is an atomic formula of the form $call_4(t_{db}, t_{rel}, t_{tid}, t_{attr}, t_{val})$, where $t_{db}, t_{rel}, t_{tid}, t_{attr}, t_{val}$ are terms of $\mathcal{L}_{fol}$ and ν is a *vaf*. then $M_{fol} \models_\nu call_4(t_{db}, t_{rel}, t_{tid}, t_{attr}, t_{val})$ iff $\langle \nu(t_{db}), \nu(t_{rel}), \nu(t_{tid}), \nu(t_{attr}), \nu(t_{val})\rangle \in \mathcal{I}_p(call_4)$. (Similarly, for atoms of depth < 4).

2. If $\mathcal{A}$ is a wff involving connectives and/or quantifiers, its satisfaction is defined in the usual inductive manner.

Theorem 2.4.1 *(Encoding Theorem)*
Let $\mathcal{A}$ be a SchemaLog formula, M_s be a SchemaLog structure, and ν a vaf. *Let encode($\mathcal{A}$) be the first-order formula corresponding to $\mathcal{A}$ and encode(M_s) be the corresponding structure for the first-order language $\mathcal{L}_{fol}$.*
Then, $M_s \models_\nu \mathcal{A}$ iff encode(M_s) $\models_\nu$ encode($\mathcal{A}$).

PROOF. Let $M_s = < U, \mathcal{I}, \mathcal{I}_{fun}, \mathcal{F} >$ be the SchemaLog structure. Let encode (M_s) be the structure $M_{fol} = < U, \mathcal{I}_f, \mathcal{I}_p >$. We shall show by induction on the structure of the formulas $\mathcal{A}$ of $\mathcal{L}$ that $M_s \models_\nu \mathcal{A}$ iff $M_{fol} \models_\nu$ encode($\mathcal{A}$).

Base case: $\mathcal{A}$ is an atom. Actually, there are four cases to consider, depending on the depth of the atom. We shall give the proof for the case when depth is 4. The proof for atoms of depth less than four is analogous. Let $\mathcal{A}$ be the atom $t_{db} :: t_{rel}[t_{tid} : t_{attr} \rightarrow t_{val}]$, where $t_{db}, t_{rel}, t_{tid}, t_{attr}, t_{val}$ are terms of $\mathcal{L}$. Now, $M_s \models_\nu t_{db} :: t_{rel}[t_{tid} : t_{attr} \rightarrow t_{val}]$, iff $\mathcal{F}(\nu(t_{db}))(\nu(t_{rel}))(\nu(t_{tid}))(\nu(t_{attr}))$ is defined in M_s, and $\mathcal{F}(\nu(t_{db}))(\nu(t_{rel}))(\nu(t_{tid}))(\nu(t_{attr})) = \nu(t_{val})$,
iff $< \nu(t_{db}), \nu(t_{rel}), \nu(t_{attr}), \nu(t_{tid}), \nu(t_{val}) > \in \mathcal{I}_p(call_4)$,
iff $M_{fol} \models_\nu call_4(t_{db}, t_{rel}, t_{tid}, t_{attr}, t_{val})$, iff $M_{fol} \models_\nu$ encode($\mathcal{A}$).

Induction: Suppose $\mathcal{A}$ is a compound formula involving connectives and/or quantifiers. We shall indicate the proof for one case; the remaining cases will follow analogously.

Let $\mathcal{A}$ be of the form $\mathcal{B} \vee \mathcal{C}$ where $\mathcal{B}$ and $\mathcal{C}$ are arbitrary SchemaLog formulas.

$$\begin{aligned}
M_s \models_\nu \mathcal{B} \vee \mathcal{C} \quad &\Leftrightarrow \quad M_s \models_\nu \mathcal{B} \text{ or } \mathrm{M} \models_\nu \mathcal{C} \\
&\Leftrightarrow \quad M_{fol} \models_\nu \text{encode}(\mathcal{B}) \text{ or } M_{fol} \models_\nu \text{encode}(\mathcal{C}) \\
&\Leftrightarrow \quad M_{fol} \models_\nu (\text{encode}(\mathcal{B}) \vee \text{encode}(\mathcal{C})\ ) \\
&\Leftrightarrow \quad M_{fol} \models_\nu \text{encode}(\mathcal{B} \vee \mathcal{C}) \\
&\Leftrightarrow \quad M_{fol} \models_\nu \text{encode } (\mathcal{A}).
\end{aligned}$$

□

From Theorem 2.4.1, with simple induction, it follows that every SchemaLog program P can be encoded into a first-order logic program $encode(P)$, such that for every SchemaLog structure M_{in}, P maps M_{in} to an output structure M_{out} iff $encode(P)$ maps $encode(M_{in})$ to $encode(M_{out})$. In simple words, this means that for all mappings between SchemaLog structures expressible by SchemaLog programs there exist corresponding transformations on the encodings of the SchemaLog structures, which are expressible as first-order logic programs. Thus, *technically* SchemaLog has no more expressive power than first-order logic as a database programming language. As a consequence of the first-order semantics, the results of axiomatizability, decidability, and compactness accrue for SchemaLog.

Discussion on Expressive Power

The results of the preceding section indicate that SchemaLog has no more expressive power than first-order logic, in view of the fact that the former can be *simulated* in the latter. This raises the question - "*What good is SchemaLog, if it does not yield a higher expressive power than first-order logic?*". To understand this question in perspective, note that the simulation of SchemaLog in first-order logic crucially relies on the assumption that a federation of conventional databases is available in *reduced form*, *i.e.* in the form of the four *call* relations - $call_1, call_2, call_3, call_4$ (see the proof of Theorem 2.4.1). The equivalence in expressive power between first-order logic and SchemaLog thus holds *only* when the former is given a federation of databases in reduced form as input, while the latter is used against databases in conventional form. Thus, notice that the *simulated* and *simulating* languages do *not* take the *same* federation of databases as input, although the inputs are *equivalent*.

Ross [Ros92] addresses a similar issue in the context of an algebra he proposes for HiLog and introduces the notion of a *relation preserving simulation*. He defines a simulation to be relation preserving, if the *simulated* as well as the *simulating* formalisms operate on the *same* database. In the context of interoperability, we can extend this notion to the level of a federation and speak of simulations that *preserve schemas*.

Definition 2.4.1 *Let $\tau : \mathcal{I}_{in} \rightarrow \mathcal{I}_{out}$ be a transformation between a class of input and output instances, and L be any logic language. We say that a program P in L* expresses *τ provided $\forall I \in \mathcal{I}_{in}, P(I) = \tau(I)$.*

Definition 2.4.2 *(Schema preserving simulation) A language L can be* simulated in a schema preserving manner *in another language L′ provided for every program P in L that expresses a transformation $\tau : \mathcal{I}_{in} \rightarrow \mathcal{I}_{out}$, there is a program P′ in L′ that expresses τ.*

A crucial point to observe in the above definition is that programs in both the simulated (L) and simulating (L') languages manipulate input instances with *identical schemas* (and hence identical data). This is to be contrasted with the kind of simulation entailed by Theorem 2.4.1, where SchemaLog programs manipulate relational databases in their conventional form, while the simulating first-order logic programs manipulate the encoded versions of these databases, which clearly have a different schema.

The theoretical motivation for schema preservation arises from the fact that if the databases in the federation are encoded arbitrarily for the purpose of simulation, useful information such as normal forms and integrity constraints would be lost. This is certainly the case with the reduced form encoding used in the proof of Theorem 2.4.1. From a practical perspective, because of autonomy requirements and also due to the prohibitive cost involved, encoding the databases in a federation into reduced form is infeasible. While Theorem 2.4.1 does not yield a schema preserving simulation, it does not *establish* that no such simulation is possible. The following theorem settles this issue with finality.

Theorem 2.4.2 *First-order logic cannot simulate SchemaLog in a schema preserving manner.*

PROOF. Consider the SchemaLog program P:

$$db' :: rel'[X \rightarrow Y] \longleftarrow db :: rel[a \rightarrow X,\ b \rightarrow Y].$$

Clearly, P generates a relation whose width is dependent on the data in *rel*. On the other hand, every relational algebraic operator produces an output with a schema that is data-independent. By induction, any first-order logic program has this property and hence the transformation expressed by P cannot be expressed by any first-order logic program. □

Theorem 2.4.2, together with Theorem 2.4.1, implies that first-order logic cannot express the mapping between a conventional database and its reduced form. On the other hand, SchemaLog can readily express this and more powerful forms of *restructuring* of databases (also see Section 2.6.2). In view of the above discussions, we see that schema preservation is an essential practical requirement for query languages for interoperability. *We conclude that under the requirement of schema preservation, SchemaLog has a strictly higher expressive power than first-order logic.*

As a final note, we remark that a language with higher expressive power under the requirement of schema preservation, leads to queries in the chosen application domain which are more natural and concise compared to the language which can only simulate the former via encodings that do not preserve schemas.

2.5 Programming in SchemaLog

For the purposes of database programming, in Section 2.5.1, we focus on the definite clause fragment of SchemaLog. We develop the fixpoint and model theoretic semantics of this fragment and establish their equivalence. In Section 2.5.2, we develop a sound and complete proof theory for the *full logic* of SchemaLog. For simplicity of exposition, we do not address the issue of equality in Sections 2.5.1 and 2.5.2. In Appendix A.1, we show how the results of these sections can be lifted to the case where equality theory is addressed.

2.5.1 Fixpoint Semantics

We will consider a program P to be a set of definite clauses. The notions of Herbrand base, Herbrand interpretation and Herbrand model follow those of the conventional ones with extensions induced by the nested structure of SchemaLog atoms.

Definition 2.5.1 *Let A be an atomic formula of depth n, $1 \leq n \leq 4$. The* **restriction of** A **to depth** m, $m < n$, *is the formula A' obtained by retaining the first m non-id components of A. When the depth is not important, we simply say that A' is a restriction of A. The restriction of an atom A of depth n to depth n, is itself.*

Example 2.5.1 *The restriction of $t_1 :: t_2[t_4 : t_3 \rightarrow t_5]$ to depth 3, 2, 1 are $t_1 :: t_2[t_3]$, $t_1 :: t_2$ and t_1 respectively.*

Definition 2.5.2 *Let I be a set of ground atoms. Then the* **closure** *of I, denoted $cl(I)$, is defined as $cl(I) = \{A \mid \exists B \in I$ s.t. A is the restriction of B to depth m, for some $1 \leq m \leq depth(B)\}$. A set of atoms I is closed if $cl(I) = I$.*

We extend the notion of closure to a set $\mathcal{I}$ of sets of atoms by defining $cl(\mathcal{I}) =_{def} \{cl(I) \mid I \in \mathcal{I}\}$.

Let P be a definite program. Then the **Herbrand universe** of P is the set of all ground (*i.e.* variable-free) terms that can be constructed using the symbols in P. The **Herbrand base** B_P of P is the set of all ground atoms that can be formed using the logical symbols appearing in P. Note that by definition, the Herbrand base of a program is closed. A **Herbrand interpretation** I of P is any closed subset of B_P. It can be shown that a Herbrand interpretation obtained from first principles using the definition of a structure by interpreting all logical symbols as themselves and the function symbols in $\mathcal{G}$ in the usual "Herbrand" style is equivalent to the above simpler notion of Herbrand interpretation. I is a **model** of P if it satisfies all the clauses in P. It is easy to show that the union (intersection) of closed subsets of B_P is closed. Then $cl(2^{B_P})$, the set of all Herbrand interpretations of P, is a complete lattice under the partial order of set inclusion $\subseteq$. The top element of this lattice is B_P and the bottom element is Φ. Union and intersection correspond to the join and meet as usual.

Definition 2.5.3 *Let P be a definite program. The mapping $T_P : cl(2^{B_P}) \rightarrow cl(2^{B_P})$ is defined as follows. Let I be a Herbrand interpretation. Then $T_P(I) = cl(\{A \in B_P \mid A \longleftarrow A_1, \ldots, A_n$ is a ground instance of a clause in P and $\{A_1, \ldots, A_n\} \subseteq I\})$.*

We have the following results.

Lemma 2.5.1 *Let P be a definite program. The mapping T_P is continuous (and hence monotone).*

PROOF. Let X be a directed[2] subset of $cl(2^{B_P})$. $\{A_1, \ldots, A_n\} \subseteq lub(X)$ iff $\{A_1, \ldots, A_n\} \subseteq I$, for some $I \in X$. To show T_P is continuous, we have to show $T_P(lub(X)) = lub(T_P(X))$, for each directed subset X. Thus,

$$\begin{aligned} A \in T_P(lub(X)) \;&\Leftrightarrow\; B \leftarrow A_1, \ldots, A_n \text{ is a ground instance of a clause in } P\ ,\\ &\qquad \{A_1, \ldots, A_n\} \subseteq lub(X), \text{ and } A \text{ is a restriction of } B.\\ &\Leftrightarrow\; B \leftarrow A_1, \ldots, A_n \text{ is a ground instance of a clause in } P \text{ and}\\ &\qquad \{A_1, \ldots, A_n\} \subseteq I, \text{ for some } I \in X, \text{ and } A \text{ is a restriction of } B.\\ &\Leftrightarrow\; A \in T_P(I), \text{ for some } I \in X,\\ &\Leftrightarrow\; A \in lub(T_P(X)). \end{aligned}$$

This proves that T_P is continuous. Monotonicity follows from this. □

Lemma 2.5.2 *Let P be a definite program and I be a Herbrand interpretation of P. Then I is a model for P iff $T_P(I) \subseteq I$.*

PROOF. I is a model for P iff for each ground instance $A \longleftarrow A_1, \ldots, A_n$ of each clause in P, $\{A_1, \ldots, A_n\} \subseteq I$ implies A $\in$ I. This is true if and only if $T_P(I) \subseteq I$.

[2] X is directed if every finite subset of X has an upper bound in X.

In particular, notice that every atom $B \in T_P(I)$ which is a restriction of A, where $A \longleftarrow A_1, ..., A_n$ is a ground instance of a clause in P and $\{A_1, ..., A_n\} \subseteq I$, is also in I (as I is closed). □

Theorem 2.5.1 *(Fixpoint characterization of Least Herbrand Model)*
Let P be a definite program. Let $\mathcal{M}(P)$ be the set of all Herbrand models of P and let $\cap\mathcal{M}(P)$ be their intersection. Then $\cap\mathcal{M}(P)$ is a model of P called the least Herbrand model of P. Further $\cap\mathcal{M}(P) = lfp(T_P) = T_P \uparrow \omega = \{A \mid A \in B_P \wedge P \models A\}$.

PROOF. We know, $\cap\mathcal{M}(P)$ is $glb(I : I$ is a Herbrand model for $P)$. It follows from Lemma 2.5.2 that this is the same as $lfp(T_P)$. The theorem now follows from Lemma 2.5.1. The details are identical to those for classical logic programming ([vK76]). □

Incorporating any of the various forms of negation studied in logic programming (*e.g.*, see [She88]) in SchemaLog is not very difficult.

2.5.2 Proof Theory of SchemaLog

In this section, we develop a sound and complete proof theory for SchemaLog as a *full-fledged logic.* We consider *arbitrary* SchemaLog theories, not just definite clauses. Analogous to first order logic, we can show that arbitrary theories can be transformed into clausal theories. This is achieved through Skolemization, as usual. We then develop the proof theory for SchemaLog theories consisting of clauses, based on resolution.

Skolemization in SchemaLog

A sentence σ in SchemaLog can be transformed into an equivalent sentence σ' in **prenex normal form**. A sentence is in prenex normal form if it is of the form $(Q_1X_1)\ldots(Q_nX_n)(F)$ where every $(Q_iX_i), i = 1, \ldots, n$, is either $(\forall X_i)$ or $(\exists X_i)$, and F is a formula containing no quantifiers. This transformation is along the lines of the one used in predicate calculus. An algorithm for this transformation can be found in Chang and Lee [CL73] and can be easily adapted for SchemaLog.

Skolemization is the process of eliminating the existential quantifiers in a formula by replacing them with suitable functions (called Skolem functions). The intuition behind Skolemization is the following. If a formula asserts that for every X, there exists a Y such that some property holds, the choice of Y could be seen as a function of

X. Skolemization simply assigns a (new) arbitrary function symbol to represent this choice function. This can be used to eliminate the existential quantifier associated with Y.

Notice that Skolemization in SchemaLog is virtually identical to that in classical first-order logic. The essential reason for this is that in SchemaLog, as in classical logic, function symbols are directly interpreted into their extensions. By contrast, HiLog (for example), interprets function symbols (as also other symbols) intensionally. There, a new symbol chosen to represent a Skolem function must be assigned a new *intension*, which may not always be possible. The authors of HiLog get around this difficulty by using an unused arity of one of the old symbols to represent the Skolem function. (See [CKW90] for the details.) For SchemaLog, since Skolemization works in a manner identical to that of predicate calculus, as explained above.

Herbrand's Theorem

By virtue of Skolemization, without loss of generality, we can restrict our attention to formulas in prenex normal form which are universally quantified. By transforming such formulas into conjunctive normal form, we can obtain SchemaLog formulas that are in *Clausal form*. Thus, such formulas may be viewed as sets of clauses. Recall the notions of Herbrand universe, Herbrand base, and Herbrand interpretations (Section 2.5.1).

Proposition 2.5.1 *Let S be a set of clauses and suppose S has a model. Then S has a Herbrand model.*

PROOF. Let I be an interpretation of S. The Herbrand interpretation I' is defined as $I' = \{A \in B_S : A \text{ is true in } I\}$. It follows by an easy induction that if I is a model of S, then I' is also a model of S. □

Lemma 2.5.3 *A set of clauses S is unsatisfiable iff it is false with respect to all Herbrand structures.*

PROOF. If S is satisfiable, then Proposition 2.5.1 shows that it has a Herbrand model. □

Following Chang and Lee [CL73], we next introduce the notion of a semantic tree. As in the classical case, we shall use semantic trees to establish the strong version of Herbrand's Theorem (see Theorem 2.5.2 below) for SchemaLog, as well as to prove the completeness of our proof procedure. The following notions are needed in defining semantic trees. Recall the notion of restriction of atoms to smaller depths (see Definition 2.5.1). The notion can be extended to literals in the obvious manner.

Definition 2.5.4 *A literal L_j is* reducible *to literal L_i, if L_i is L_j restricted to $depth(L_i)$. Let A be an atom. The literal $\neg A'$* contradicts *A, if A is reducible to A'. The set $\{A, \neg A'\}$ is called a* contradictory pair.

Notice that if $\neg A'$ contradicts A, it does not in general follow that $\neg A$ contradicts A'. An example is $A = db :: rel[attr]$ and $A' = db :: rel$.

Definition 2.5.5 *Let S be a set of clauses, and let B_S be the Herbrand base of S. A* semantic tree *for S is any tree whose edges are labeled with finite sets of ground literals such that:*
(i) Each node v has a finite number of children; let $e_1, \cdots, e_k$ be the edges connecting v to its children and let $lit(e_i)$ denote the (finite) set of literals labeling e_i. We can view each set $lit(e_i)$ also as denoting the conjunction of the literals in this set. Then, $lit(e_1) \vee \cdots \vee lit(e_k)$ is a tautology.
(ii) For each node v, the union of all labels of edges appearing in the branch from the root down to v, contains no contradictory pair of literals.

For a node v of a semantic tree, we let $I(v)$ denote the union of all labels of edges appearing in the branch from the root down to v. Note that in general $I(v)$ can be viewed as a partial interpretation.

Definition 2.5.6 *Let B_S be the Herbrand base of a set S of clauses. A semantic tree T for S is said to be* complete *provided for every leaf node v of T, and for every atom $A \in B_S$, $I(v)$ contains A or $\neg A$. Notice that a complete semantic tree can be infinite.*

Definition 2.5.7 *A node v of a semantic tree T for a set of clauses S is a* failure node *if $I(v)$ falsifies some ground instance of a clause in S, but $I(v')$ does not falsify any ground instance of a clause in S for every ancestor node v' of v. T is said to be* closed *provided every branch of T terminates at a failure node. A node v of a closed semantic tree is called an* inference node *if all the immediate descendant nodes of v are failure nodes.*

We finally state the strong version of Herbrand's Theorem for SchemaLog which plays an important role in the proof of completeness of the proof procedure. Its proof given below, is analogous to that for classical predicate calculus.

Theorem 2.5.2 *(Herbrand's Theorem) A set S of wffs in clausal form is unsatisfiable iff every complete semantic tree T for S has a finite closed subtree.*

PROOF. It has been shown in Section 2.4 that there is a transformation from SchemaLog to first order logic such that a SchemaLog formula $\mathcal{A}$ is true in a structure M_s under vaf ν iff the corresponding first order formula encode($\mathcal{A}$) is true in the corresponding first order structure encode(M_s) under the vaf ν (Theorem 2.4.1). Herbrand's theorem can now be proved from the above result using a technique similar to that used for predicate calculus [CL73]. The main observation is that whenever S is unsatisfiable, every branch of any complete semantic tree T of S must have a failure node. Since each node of T has a finite number of children, an application of König's Lemma at once implies the existence of a finite closed subtree of T. The details are straightforward and are suppressed. $\Box$

Note that just as in the classical case [CL73], we get as an easy consequence of Theorem 2.5.2 that a set S of clauses is unsatisfiable if and only if some finite subset of the ground instances of S is.

Unification

Unification in SchemaLog has to be treated differently from the way it is done conventionally. In our case, unlike in predicate calculus, there is a natural need for literals of unequal depth to be unified. To see this, consider the following example.

Consider the definite program $P = \{db :: rel[attr] \longleftarrow \}$ asserting the existence of a database db, with a relation named rel, for which an attribute $attr$ is defined. Now, consider a query : $? - db :: rel$ which asks about the existence of a database db, with a relation named rel defined on it. Resolution in the conventional sense would not result in a refutation (whereas it should!). Now let us "switch" the (head of the) rule and the goal, *i.e.* consider the program $P = \{db :: rel \longleftarrow \}$ and the query $? - db :: rel[attr]$. Intuitively, we understand that the resolution should fail.

The above example illustrates two key issues: (1) Unification in SchemaLog involves 'unlike' literals and (2) unifiability is not commutative. Intuitively, the above issues are related to the definition of closure used in the fixpoint semantics. This in turn is associated with the nested structure of atoms allowed in our language. Thus, the conventional notion of unification needs to be extended[3]. We discuss this next.

Definition 2.5.8 *A* substitution *is a finite set of the form* $\{t_1/X_1, \ldots, t_n/X_n\}$ *, where* $X_1, \ldots, X_n$ *are distinct variables, and every term* t_i *is different from* X_i, $1 \leq i \leq n$.

Definition 2.5.9 *A* unifier *of literal* L_i *to literal* L_j *is a substitution* θ *such that* $L_j\theta$ *is reducible (see Definition 2.5.4) to* $L_i\theta$. *Literal* L_i *is* unifiable to *literal* L_j *if there is a unifier of* L_i *to* L_j.

Definition 2.5.10 *A unifier* σ *of a literal* L_i *to literal* L_j *is a* most general unifier *(mgu) iff for each unifier* θ *for* L_i *to* L_j*, there exists a substitution* λ *such that* $\theta = \sigma\lambda$.

The Unification Algorithm

Our unification algorithm is essentially similar to the one for classical logic. We have to adopt certain modifications to account for the peculiar syntax of SchemaLog and the somewhat different notion of unification defined above. We develop an algorithm below by modifying the unification algorithm discussed, *e.g.*, in Ullman [Ull89a].

Consider any two SchemaLog atoms A and B. Without loss of generality, we may assume that there is no variable which occurs in both A and B. (Such variables can always be renamed). We would like to test if A can be unified to B. Clearly, a necessary condition for this is that *depth(A)* $\leq$ *depth(B)*, which we shall assume in the algorithm below.

Algorithm 2.5.1 *Computing the Most General Unifier.*

INPUT: *Atomic formulas A and B with disjoint sets of variables.*

OUTPUT: *A most general unifier of A to B or an indication that none exists.*

METHOD: The algorithm consists of two phases. Phase I distinguishes the *equivalent subexpressions* of A and B that must become identical in the MGU. Phase II determines if an MGU exists.

[3]The directionality associated with unification also arises in F-logic [KLW95] but for a different reason: a molecule with fewer components may be unified to one with more components. This feature is present in SchemaLog as well, at the molecular level. But, unlike F-logic, SchemaLog unification needs to be directional even at the atomic level.

Phase I: *Finding equivalent subexpressions.*

A tree corresponding to each of A and B is constructed first. The following rules inductively define the tree for A. (The tree for B is constructed in a similar way.)

1. If A is of the form $t_{db} :: t_{rel}[t_{tid} : t_{attr}\rightarrow t_{val}]$, then the tree for A has an unlabeled root with 5 children, $v_{db}, v_{rel}, v_{id}, v_{attr}, v_{val}$ from left to right, in that order where v_i is the root of the tree for the term t_i, $i \in \{db, rel, id, attr, val\}$ (If A is an atom of depth less than four, its tree will have an unlabeled root with children corresponding to the terms appearing in A.)

2. Let t be a term of the form $f(t_1, \ldots, t_n)$ for a function symbol f of arity n and terms $t_1, \ldots, t_n$. Then the tree for t has a root labeled f. The root has n children $v_1, \ldots, v_n$, where v_i is the root of the tree for t_i, $i = 1, \ldots, n$.

Rules 1 and 2 completely describe the tree for any SchemaLog atom.

After building the trees for A and B, we group their nodes into *equivalence classes*. These equivalence classes can be represented by the equivalence relation $\equiv$. The rules for defining $\equiv$ are:

1. If r_A and r_B are the roots of the two trees, then $r_A \equiv r_B$.

2. Suppose m, n are any nodes of t_A and t_B respectively, such that $m \equiv n$. Then two cases arise:

<u>Case 1</u>: m is the root r_A and n is the root r_B. In this case, let $u_1, \ldots, u_k$ be the children of r_A and $v_1, \ldots, v_n$ be the children of r_B, where $1 \le k \le n \le 4$. We say that a child u_i corresponds to a child v_j, provided both correspond to a database, a relation, a tuple-id, an attribute or a value term. Whenever u_i corresponds to v_j, set $u_i \equiv v_j$.

<u>Case 2</u>: m and n are both any nodes of t_A and t_B, other than their roots. In this case, m and n must be labeled. If they are both internal nodes, they must be labeled by some function symbols. If the function symbols are distinct, conclude "A cannot be unified to B" and exit. Otherwise, they must have the same number of children, say $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ respectively. Then set $u_i \equiv v_i, i = 1, \ldots, n$.

3. If nodes m and n are labeled by the same variable, then $m \equiv n$.

4. $n \equiv n$ for any node n; if $n \equiv m$, then $m \equiv n$; if $n \equiv m$ and $m \equiv p$, then $n \equiv p$.

Phase II of the algorithm constructs the MGU by considering each equivalence class obtained from the previous phase. This phase is identical to phase II of the unification algorithm for classical logic given in Ullman [Ull89a], to which we refer the reader for details. □

We give an example of unification, to illustrate the algorithm.

Example 2.5.2 *We will consider unifying the SchemaLog formula A to B where, $A = f(X, g(X)) :: Y$ and $B = f(g(U), V) :: rel(g(W))$.*

The equivalence classes we obtain are: $\{W\}$, $\{U\}$, $\{Y, rel\}$, $\{X, g(U)\}$, $\{g(X), V\}$ *and the MGU* τ *is obtained as:* $\tau(W) = W$, $\tau(U) = U$, $\tau(Y) = rel$, $\tau(X) = g(U)$, $\tau(V) = g(g(U))$. □

Theorem 2.5.3 *Unification Theorem:*
Given atomic formulas A and B, Algorithm 2.5.1 correctly computes the most general unifier of A to B if the mgu exists.

The proof of this theorem follows the same lines as the one discussed for classical logic in Ullman [Ull89a]. The modifications to the proof to account for the modified phase I of the algorithm are straightforward.

Resolution and Completeness

In this section, we show that the extension of the resolution-based proof procedure to the higher-order setting is sound and complete for SchemaLog. Before presenting resolution, we recall the following notions.

Definition 2.5.11 *Let L_i and L_j be two literals in a clause C. If there is a most general unifier σ of L_i to L_j, then $C\sigma$ is called a* factor *of C. If $C\sigma$ is a unit clause, it is called a* unit factor *of C.*

Definition 2.5.12 *Let C_1 and C_2 be two clauses (called* parent clauses*) with no variables in common. Let L_1 and L_2 be two literals in C_1 and C_2 respectively. If L_1 has a most general unifier σ to $\neg L_2$, then the clause obtained by removing $L_1\sigma$ and $L_2\sigma$ from a disjunction of $C_1\sigma$ and $C_2\sigma$ is called a* binary resolvent *of C_1 and C_2. The literals L_1 and L_2 are called the* literals resolved upon.

Definition 2.5.13 *A* resolvent *of (parent) clauses C_1 and C_2 is a binary resolvent of a factor of C_1 and a factor of C_2.*

Definition 2.5.14 *A clause C is a variant of another clause D provided there is a substitution θ which maps variables in D to distinct variables in C such that $C \equiv D\theta$. Let S be a set of clauses standardized apart in the classical sense. A* deduction *from S is a finite sequence $C_1, \ldots C_n$ of clauses such that for $i = 1, \ldots, n$, either C_i is a variant of some clause in S, or C_i is a resolvent of C_j and C_k, for some $j, k < i$.*

The proof for the following lemma, and the proof for the completeness theorem that follows, both closely follow the proofs of the corresponding results for predicate calculus [CL73]. In both cases, we provide the major steps and ideas involved in the proof; other details are analogous to those in [CL73].

Lemma 2.5.4 *Lifting Lemma:*
If C_1' and C_2' are instances of C_1 and C_2, respectively, and if C' is a resolvent of C_1' and C_2', then there is a resolvent C of C_1 and C_2 such that C' is an instance of C.

PROOF. Variables in C_1 and C_2 can be renamed such that there are no common variables in them. Let L_1' and L_2' be the literals of C_1' and C_2' (respectively) that are resolved upon and let γ be the mgu of L_1' to $\neg L_2'$. Let C' be the clause obtained by removing $L_1'\gamma$ and $L_2'\gamma$ from a disjunction of $C_1'\gamma$ and $C_2'\gamma$. There is a substitution θ such that $C_1' = C_1\theta$ and $C_2' = C_2\theta$. Let λ_i be the mgu for the literals, say $\{L_i^1, \ldots, L_i^{k_i}\}$ in C_i, which correspond to L_i'. Let $L_i \equiv L_i^1\lambda_i \equiv \cdots \equiv L_i^{k_i}\lambda_i$. Clearly, L_i is a literal in the factor $C_i\lambda_i$ of C_i. It follows from this that L_i' is an instance of L_i. Since L_1' is unifiable to $\neg L_2'$, L_1 is unifiable to $\neg L_2$. Let σ be the mgu of L_1 to $\neg L_2$.

Let C be the disjunction $D_1 \vee D_2$ where D_i is the disjunction obtained by removing $L_i\sigma$ from $(C_i\lambda)\sigma$, $i = 1, 2$. From this, it can be proved that C is a resolvent of C_1 and C_2. Clearly, C' is an instance of C since $C' = E_1 \vee E_2$, where E_i is obtained by removing $L_i'\gamma$ from $(C_i'\gamma)\sigma$, $i = 1, 2$, and $\lambda \circ \sigma$ is more general than $\theta \circ \gamma$. $\square$

Theorem 2.5.4 *Soundness and Completeness of Resolution:*
A set S of clauses is unsatisfiable if and only if there is a deduction of the empty clause $\square$ from S.

PROOF. Suppose S is unsatisfiable. Let B_S be the Herbrand base of S. Let T be a complete semantic tree for S. By Herbrand's theorem (Theorem 2.5.2), T has a subtree T', which is a finite closed semantic tree. If T' has only one (root) node, then $\square$ must be in S, giving a trivial deduction of $\square$. If T' has more than one node, T' must have at least one inference node, for otherwise we can get a trivial contradiction to the finiteness of T'. Let v be an inference node of T'. Assume without loss of generality that v has exactly two children - v_1, v_2. (By the definition of a semantic tree, v has ≥ 2 children, and the case where v has > 2 children can be handled similarly to the present case.) Clearly, v_1, v_2 are failure nodes. Let A and $\neg A$ be the labels of the edges (v, v_1) and (v, v_2) respectively. But since v is not a failure node, there must exist two ground instances C_1' and C_2' of clauses C_1 and C_2 in S such that C_1' and C_2' are false in $I(v_1)$ and $I(v_2)$ respectively, but both C_1' and C_2' are not falsified by $I(v)$. Therefore, C_1' must contain $\neg A$ and C_2' must contain A. By resolving upon the literals A and $\neg A$, we can obtain a resolvent C' of C_1' and C_2', which must be false in $I(v)$. By Lemma 2.5.4, there is a resolvent C of C_1 and C_2 such that C' is a ground instance of C. Let T'' be the closed semantic tree for $(S \cup \{C\})$, obtained from T' by deleting all nodes and edges below the first node from the root down where C' is falsified. Clearly, the number of nodes in T'' is strictly fewer than that in T'. Since T' and hence T'' is finite, we can apply this technique inductively by adding resolvents of clauses in $S \cup \{C\}$ (obtained by deduction) to $S \cup \{C\}$ and so forth, eventually obtaining the empty subtree consisting of only the root. At this point we would clearly have obtained a deduction of the empty clause $\Box$ from S.

Soundness follows in a straightforward way. $\Box$

Molecular programming vs Atomic programming

We mentioned in Section 2.3.1 that molecular formulas can be introduced in the syntax of SchemaLog as an abbreviation for a conjunction of atomic formulas. Molecular formulas can indeed provide a mechanism for direct, convenient programming. Let us illustrate this point with an example.

Consider the (good old!) example of grandfathers. The grandfather predicate can be defined (from the parent predicate) in SchemaLog using the rule

$$db :: grandpa[f(X, Y) : pers \rightarrow X,\ grndFath \rightarrow Y] \longleftarrow$$
$$db :: par[T_1 : pers \rightarrow X,\ fath \rightarrow Z],\ db :: par[T_2 : pers \rightarrow Z,\ fath \rightarrow Y].$$

Notice that this rule makes use of molecules. The precise model-theoretic semantics of molecular formulas in SchemaLog relies on their equivalence to a corresponding conjunction of atoms. However, as the reader can very well verify, expressing the same rule using only atoms[4] would be quite cumbersome. We remark that in a relational context, one could completely dispense with tuple-id's (in an interface) as long as molecular programming is supported by the system. The system can always fill in the id's. The point, however, is that tuple-id's are needed in order to keep the model-theoretic semantics of SchemaLog simple, in that they allow references to tuples via

[4]This will necessitate two rules - one for each argument of the predicate *grandpa*.

their intensions (tuple-id's) as opposed to their extension (i.e. the actual tuple of values). Besides, they are quite in keeping with our eventual objective of providing for the integration of disparate data models, including the object-oriented model. We remark that the fixpoint theory and proof theory of molecular programs are straightforward extensions of those for atomic programs. In the rest of this chapter, we shall freely make use of molecules in our examples. While the use of molecules makes the programming of certain queries easier, we shall see later (Section 2.6) that clever manipulation of tuple id's gives SchemaLog great power in expressing sophisticated queries, *even in* the relational context.

Programming Predicates

In the context of queries as well as view definitions, it will be convenient to have the (facility for) predicates (which are not part of any database) available. The difference between such predicates and those in a database is that they may be regarded as corresponding to temporary tables and hence one need not carry along the tuple-id's with such predicates. We call such predicates **programming predicates** (for distinction from the database predicates).[5] On the technical side, programming predicates can be easily incorporated in SchemaLog by introducing a separate set of predicate symbols and then interpreting them "classically". We shall freely make use of programming predicates in the examples of Section 2.6 (*e.g.*, see query Q_4 in Section 2.6.1). The main distinction between programming predicates and database predicates is that unlike database predicates, the schema components of programming predicates do not have a formal status in SchemaLog. Thus, programming predicates have a syntax similar to predicates in Datalog.

2.6 Applications of SchemaLog

In this section, we give a variety of examples illustrating the power and applicability of SchemaLog for database programming, schema integration, schema evolution, cooperative query answering, and aggregation. We also make a case for adopting a

[5]For programming predicates we use the conventional syntax $\langle \textit{pred-name}\rangle(\langle arg_1\rangle, \ldots, \langle arg_n\rangle)$. Note that this introduces ambiguity in the syntax of SchemaLog, as a programming predicate could now be confused with a functional term! We can remove this ambiguity by requiring functional terms to conform to the syntax $f < t_1, \ldots, t_m >$. For the sake of clarity and simplicity of exposition, we ignore this point. The intended meaning of SchemaLog expressions will always be clear from the context.

uniform framework for schema integration and evolution and illustrate via examples how SchemaLog could fulfill this need.

2.6.1 Database Programming and Schema Browsing

The main advantage of SchemaLog for database programming lies in its simplicity of syntax which buys it ease of programming. Yet its higher-order syntax gives it sufficient power to express complex queries in a natural way thus bringing programming closer to intuition. For instance let us take a look at the following example query adopted from [CKW90].

(Q_3) *"Find the names of all the binary relations in which the token 'john' appears."*

This query can be expressed in HiLog, the following way[6]:

$relations(Y)(X) \longleftarrow X(Y, Z)$

$relations(Z)(X) \longleftarrow X(Y, Z)$

$? - relations(john)(X)$

Now, consider a variant of Q_3

(Q_4) *"Find the names of* all *the relations in which the token 'john' appears."*

It seems the only way such a query could be expressed in HiLog is by writing one set of rules for each arity of the various relations present in the database (this presupposes the user's knowledge of the schema of the database). By contrast, in SchemaLog this query can be expressed quite elegantly, as follows.

$relations(X, Rel) \longleftarrow db :: Rel[I : A \rightarrow X]$

$? - relations('john', Rel)$

Here, *relations* is the programming relation that contains information on each token in database *db* and the name(s) of the relation(s) in which the token appears. Note that we have considered the query in the context of just one database. If all databases and relations where '*john*' occurs are of interest, we could write the rule

$whereabouts(X, DB, Rel) \longleftarrow DB :: Rel[I : A \rightarrow X]$, and ask the query,

$? - whereabouts('john', DB, Rel)$.

On the other hand, if we *specifically* want the binary relations in which '*john*' appears (query Q_3), the expression of this query would be less direct (and concise) in SchemaLog than in HiLog, in that the SchemaLog query would have to use (stratified) negation:

[6]Incidentally, the same browsing capability is available in F-logic too.

$$arity_{\geq 2}(Rel) \longleftarrow db :: Rel[A, B],\ A \neq B.$$
$$arity_{>2}(Rel) \longleftarrow db :: Rel[A, B, C],\ A \neq B,\ B \neq C,\ A \neq C.$$
$$arity_{2}(Rel) \longleftarrow arity_{\geq 2}(Rel),\ \neg arity_{>2}(Rel).$$
$$binary\_where(X, Rel) \longleftarrow db :: Rel[I : A \rightarrow X],\ arity_2(Rel).$$
$$?-binary\_where('john', Rel).$$

We leave it to the reader to judge which of the two types of queries Q_3 and Q_4 above is more "typical" and practically useful. Furthermore, in Section 2.6.5, we revisit query Q_3 and illustrate how SchemaLog extended with aggregate functions can express this query in a concise way (see Example 2.6.4).

Next, we present another interesting program that demonstrates the usefulness of SchemaLog for database programming. *Natural join* is a ubiquitous operation in database applications. This program demonstrates how SchemaLog could be used to invoke natural join in unconventional, but practically useful settings. Consider the query

(Q_5) *"Given two relations r and s (in database db), whose schemas are unknown, compute their natural join."*

It is obvious that this query cannot be expressed in classical logic. In SchemaLog, this query can be expressed as follows.

$$db :: join(r.s)[f(U, V) : A \rightarrow X] \longleftarrow db :: r[U : A \rightarrow X],\ db :: s[V : B \rightarrow Y],\ \neg nonJoinable(U, V).$$
$$db :: join(r.s)[f(U, V) : A \rightarrow X] \longleftarrow db :: r[U : B \rightarrow Y],\ db :: s[V : A \rightarrow X],\ \neg nonJoinable(U, V).$$
$$nonJoinable(U, V) \longleftarrow db :: r[U : A \rightarrow X],\ db :: s[V : A \rightarrow Y],\ X \neq Y.$$

In this program, a pair of tuples u, v from relations r and s respectively, is regarded *nonJoinable* if r and s have a common attribute *attr* on which u and v disagree (rule 3). In all other cases, they are regarded joinable. The join rules copy all components from a pair of joinable tuples. For each tuple in the result relation, the sub-tuple corresponding to relation r is computed in rule 1. Rule 2 computes the sub-tuple corresponding to relation s. Since the tuples are joinable, they can be safely copied componentwise without fear of inconsistency. This example also demonstrates how tuple-id's can be used to write powerful yet elegant SchemaLog programs. Sections 2.6.4 and 2.6.5 contain more examples of the use of tuple-id's in other contexts.

2.6.2 Schema Integration

One of the requirements for schema integration in an MDBS is developing a unified representation of semantically similar information structured and stored differently in the individual component databases. The concept of *mediator* was proposed by Wiederhold [Wie92] as a means for integrating data from heterogeneous sources. The expressive power of SchemaLog and its ability to resolve data/meta-data conflicts suggests that it has the potential for being used as a platform for developing mediators. We illustrate below, how SchemaLog's higher-order syntax can be used to achieve this in the case where the component databases are relational.

Consider the examples in Section 2.2. It might be argued that in order for an end user to use the language for querying databases belonging to a federation, she has to be aware of the schemas belonging to the individual databases she is interested in. The queries discussed in Section 2.2 are only for illustrating the power of the language. The idea is to use SchemaLog as a vehicle for formulating higher-order views over the databases so that the user can interact with an interface which is transparent to the differences in the component database schemas.

For instance, consider the following example[7] of higher-order view defined over the university federation of Example 1.1.1.

$db\text{-}view :: p[f(D, C, S, univ\_A) : department {\rightarrow} D,\ categ {\rightarrow} C,\ a\_sal {\rightarrow} S,\ db {\rightarrow} univ\_A]$
$\quad \longleftarrow\ univ\_A :: pay\_info[T : category {\rightarrow} C,\ dept {\rightarrow} D,\ avg\_sal {\rightarrow} S].$

$db\text{-}view :: p[f(D, C, S, univ\_B) : department {\rightarrow} D,\ categ {\rightarrow} C,\ a\_sal {\rightarrow} S,\ db {\rightarrow} univ\_B]$
$\quad \longleftarrow\ univ\_B :: pay\_info[T : category {\rightarrow} C,\ D {\rightarrow} S],\ D \neq category.$

$db\text{-}view :: p[f(D, C, S, univ\_C) : department {\rightarrow} D,\ categ {\rightarrow} C,\ a\_sal {\rightarrow} S,\ db {\rightarrow} univ\_C]$
$\quad \longleftarrow\ univ\_C :: D[T : category {\rightarrow} C,\ avg\_sal {\rightarrow} S].$

In this example, the (view) relation p is placed in a unified (derived) database called *db-view*. Here, p provides a unified view of all component databases. This illustrates the use of rules for defining views. The idea is that a logic program can define a unified view of different schemas in a MDBS, which can be conveniently queried by a federation user. The use of logic rules offers great flexibility in setting up such views. In like manner, a component database can be structured using SchemaLog to conform to the schema of another database.

This approach to unifying representations in component databases obviates the

[7]This example is an adaptation of a similar example in [KN88].

need for a canonical datamodel (see Section 1). In fact, in contrast with the CDM-based approach, this approach affords great flexibility for maintaining mappings against changes to component representations. In recent work, Turini et. al. ([ART96, GD96]) at the University of Pisa have implemented a mediator language using SchemaLog.

2.6.3 Schema Evolution

Schema Evolution is the process of assisting and maintaining the changes to the schematic information and contents of a database. It is a somewhat abused term in the database field, in that it has been interpreted to mean different things by different researchers. While Kim [Kim90] treats versioning of schema for object management as schema evolution, Nguyen and Rieu [NR89] considers the various schema change operations and the associated consequences as being its main issues. Osborn [Osb89] gives some interesting perspectives on the consequences of the polymorphic constructs in object-oriented databases and how this aids in avoiding code 'evolution'.

An important issue in schema evolution is to provide *evolution transparency* to the users, whereby they would be able to pose queries to the database based on a (possibly old) version of the schema they are familiar with, even if the schema has evolved to a different state. In related work, Ullman [Ull87] argues for the need for allowing the user to be ignorant about the structure of the database and pose queries to the database with only the knowledge about the attributes (in all relations) of the database. This will make the front-end to the user more declarative, as she is no longer bothered about the details of the database schema. As pointed out by Ullman, all natural language interfaces essentially require a facility to handle such needs.

Consider an application which has schema changes happening in a dynamic way. Every time the schema gets modified, the previous application programs written for the database become invalid and the user will have to rewrite/modify them after 'updating' herself about the schema status. We maintain that a end user should not be bothered with the details about the schema of the database she is using, especially if it keeps changing often. A better approach would be to assume that the user has the knowledge of a particular schema and let her use this to formulate queries against the database, even after the schema has been modified. The idea is to shield the modifications to the schema of the database from the user as much as possible. As a consequence, it should be possible to maintain currency and relevance of application programs with very little modifications to account for the changes to the schema.

We argue that a uniform approach to schema integration and evolution is both desirable and possible. We view the schema evolution problem from the schema integration point of view in the following way. Each stage of the schema evolution may be conceptually considered a different (database) schema that we are dealing with. The mappings between different database schemas can be defined using logic programs in a suitable higher-order language such as SchemaLog. This framework affords the possibility of *schema-independent querying and programming.*

We consider an example to illustrate our approach. This example assumes there has been no loss of information in the meta-data, between different stages of the evolution.

<u>Time t1</u>:

$schema_1$: $rel_1(a_{11}, a_{12}, a_{13})$ $rel_2(a_{21}, a_{22})$.

<u>Time t2 (current schema)</u>:

$schema_2$: $rel_1(a_{11}, a_{12})$ $rel'_1(a_{12}, a_{13})$ $rel_2(a_{21}, a_{22})$.

Relation rel_1 has been split into rel_1 and rel'_1 at time t2 (assuming the decomposition is loss-less join).

The following SchemaLog program defines a mapping between the two schemas.

$schema_1 :: rel_1[f(X, Y, Z) : a_{11} \rightarrow X,\ a_{12} \rightarrow Y,\ a_{13} \rightarrow Z] \longleftarrow$
$\quad schema_2 :: rel_1[I' : a_{11} \rightarrow X,\ a_{12} \rightarrow Y],\ schema_2 :: rel'_1[I'' : a_{12} \rightarrow Y,\ a_{13} \rightarrow Z]$
$schema_1 :: rel_2[f(X, Y) : a_{21} \rightarrow X,\ a_{22} \rightarrow Y] \longleftarrow schema_2 :: rel_2[I' : a_{21} \rightarrow X,\ a_{22} \rightarrow Y]$

Suppose the user has a view of $schema_1$: she can still pose queries with that view. The transformation program will take care of the relevant evolutionary relationship between the two schemas. Besides, since the mapping between older versions and evolved versions of the schema is maintained declaratively as a logic program, the maintenance of application programs becomes much easier.

One complication that may arise in the context of schema evolution is that evolution might involve some loss of (meta-)information (say deletion of attributes). How can we produce meaningful answers to queries (based on an older version of the schema) which refer to such "lost" information? We suggest a *cooperative query answering* approach to this problem in the following section.

2.6.4 Cooperative Query Answering

Research in the area of cooperative query answering (CQA) for databases seeks to provide relevant responses to queries posed by users in cases where a direct answer is not very helpful or informative. An overview of the work done in this area can be found in Gaasterland *et. al* [GGM92]. We also consider the aspect of CQA, concerned with answering queries in data/knowledge-base systems by extending the scope of the query so that more information can be gathered in the answers, as discussed in Cuppens and Demolombe [CD88]. Responses can be generated by looking for details that are related to the original answers, but are not themselves literal answers of the original query.

Consider the application of schema evolution discussed in the previous section. We mentioned that in the case of evolution involving loss of meta-information, for a query that addresses the 'lost' meta-information, one should not just return a direct nil/false answer, but should provide more relevant information pertaining to the query. This cooperative functionality can be realized in SchemaLog as the following example illustrates.

Example 2.6.1 *Suppose we want to capture parts of an old schema, that are discontinued in a new one. Note that values in one database might well correspond to parts of the schema in the other.* [8]

The following SchemaLog program computes the "discontinued" parts of a schema.

$items(Schema, R) \longleftarrow Schema :: R$
$items(Schema, A) \longleftarrow Schema :: R[A]$
$items(Schema, V) \longleftarrow Schema :: R[I : A \rightarrow V]$
$discont(S_{new}, X) \longleftarrow items(S_{old}, X), \neg items(S_{new}, X).$

Here $\neg$ is just stratified negation. First, *items* pairs up schemas and the various items of information that exists in them: relation names, attribute names, and their values. Then *discont* simply says X is an item that is discontinued from the database. Embellishments can be easily made to this basic idea if information on when certain item of (meta-)information was deleted or discontinued were to be kept. In such cases, in addition to telling the user "this item no longer exists in the current database" we

[8]Notice that the issue of having "correspondence tables" or mappings between old names and new ones as commonly arises in actual implementation and maintenance of federations can be suppressed without loss of generality, because such tables would simply add some edb relations to a logic program that maps the old database to the new one.

can also tell them when it was dropped. A very similar approach can also be taken for identifying items which are newly introduced in S_{new} which never existed in S_{old}.

The second aspect of CQA of interest to us arises when we want to generalize responses to queries. but it is different from the earlier approach in many ways, as the following example illustrates. This example will also illustrate a very useful and powerful way of querying (also involving schema browsing).

Example 2.6.2 *Consider the query*

(Q_6) *"Tell me all about 'john' that you can possibly find out from the database."*

For simplicity, suppose 'john' is a token (i.e. it is only a value) in the database we are considering. The following program expresses this query (T is the token of interest).

(1) $interest(T, R, I, A, V) \longleftarrow db :: R[I : X \rightarrow T,\ A \rightarrow V]$.

(2) $interest(T, R, I, A, V) \longleftarrow interest(T, S, J, B, U),\ db :: R[I : X \rightarrow U,\ A \rightarrow V]$.

(3) $info(T, R, A, V) \longleftarrow interest(T, R, I, A, V)$.

Rule (1) says if token T occurs as a value of attribute X for tuple I in relation R, then the 5-tuple $\langle T, R, I, A, V\rangle$, where V is any (other) value in the tuple where T occurs, and A its attribute name, is of interest. The second rule says that if a certain token U is of relevance to T, then all 5-tuples that are interesting with respect to U, are of interest to T. Rule (3) simply collects tuples of T, R, A, V where T is a token. R is a relation name. A is an attribute name and V is a value (of the attribute A) which pertain to token T.

Now, (under the simple assumption that all information about '*john*' is contained in a single database). the query Q_6 can be expressed as

$? - info('john', R, A, V)$.

In order to make the response for the above query much more meaningful to the user, we can add the following rule to the program.

(4) $schema\_rich\_info(T) :: R[I : A \rightarrow V] \longleftarrow interest(T, R, I, A, V)$.

This rule generates a set of databases, each corresponding to a token that appears in the input database. Each such database has relations containing those tuples in the corresponding relation in the input database, which pertain to the token directly or indirectly.

As a related query, one might want to verify whether two individuals, say '*john*' and '*mary*', in a database are related. Indeed, one might even want to know how they are related. The idea is that '*john*' and '*mary*' are considered related if they both appear in the same tuple in some relation, where the relation is an existing database relation or is obtained via a sequence of equijoins from existing relations. In addition, the output should also include the details of the equijoin and the schema information that is essential to the relationship between '*john*' and '*mary*'. The challenge is to express this query without a detailed knowledge of the schema of the database. In SchemaLog, this can be readily expressed as

$$
\begin{aligned}
&db_{new} :: interest[X \rightarrow T,\ relnship(R, A) \rightarrow V] \longleftarrow\ db :: R[I : X \rightarrow T,\ A \rightarrow V].\\
&db_{new} :: interest[X \rightarrow T,\ relnship(equijoin(P, R, B, C), A) \rightarrow V] \longleftarrow\\
&\quad db_{new} :: interest[X \rightarrow T,\ relnship(P, B) \rightarrow U],\ db :: R[I : C \rightarrow U,\ A \rightarrow V],\ \neg in(R, P).
\end{aligned}
$$

db is the existing database while db_{new} is the newly created one. The membership test, performed using predicate *in* makes sure no self joins are performed, and so the computation terminates. It is straightforward to write rules to define predicate *in*. On the other hand, for performance reasons, one may even want to implement *in* as a "built-in" predicate. The relationship between '*john*' and '*mary*' can now be queried as

$? - db_{new} :: interest[X \rightarrow \text{'}john\text{'},\ R \rightarrow \text{'}mary\text{'}]$.

In a more complex situation where an item is not known to be a token (*i.e.* it could be an attribute, relation, or value), one can easily write appropriate rules in SchemaLog to browse/navigate through the schema and compile the relevant information. We close this section noting that CQA (together with schema browsing/navigation) does indeed find interesting applications in the context of a federation. E.g., '*john*' could be an international criminal (!) on whom information may have to be tracked down from a (criminal) MDBS operated by *Interpol*. The point is that SchemaLog is well equipped to handle such situations. The (inevitably) numerous aliases of '*john*' could be captured as an edb relation representing the correspondence mappings between names across the component databases of the federation.

2.6.5 Aggregation

Aggregate functions constitute an important functionality in practical database query languages. So far, our discussions and examples illustrating the expressive power of

SchemaLog have mainly drawn upon its higher-order features. In this section, we informally discuss SchemaLog extended with aggregate functions. We shall show how a clever manipulation of tuple-id's can be used to express powerful aggregate computations. Normally, aggregate queries considered in the literature as well as those implemented by commercial systems involve collecting the (multisets of) values appearing in a column (or more), grouped according to specific criteria, and then applying any of the system-supplied aggregate operations - **avg**, **count**, **max**, **min**, **sum**. The crucial point is that values are retrieved from individual columns, one column at a time. We call such conventional aggregation *vertical aggregation* for convenience. We shall see that not only is it possible to express the conventional forms of grouping as in SQL, SchemaLog can express even novel (and practically useful) forms of grouping (and hence aggregation) which have no counterparts in SQL. Throughout this section, we shall mainly consider aggregate queries in the context of non-recursive queries. The semantics of aggregate queries in deductive databases (with and without recursion) is discussed in Ramakrishnan et al [MPR90]. Based on this theme, the semantics of SchemaLog queries with aggregates (without recursion) can be obtained as follows. A SchemaLog rule with aggregates is of the form

$$db :: rel[tid : attr_1 \rightarrow val_1, \ \ldots, \ attr_k \rightarrow val_k, \ aggAttr_1 \rightarrow agg_1(X_1), \ \ldots,$$
$$aggAttr_n \rightarrow agg_n(X_n)] \ \longleftarrow \ \langle expression \rangle.$$

Here, $tid, attr_i, aggAttr_j, val_k$ are terms as usual; agg_i are one of the usual aggregate functions. The $\langle expression \rangle$ is a conjunction of any usual SchemaLog molecules, and programming and built-in predicates. The grouping is captured by the use of the tuple-id tid in conjunction with the attribute names $aggAttr_i$. Suppose db and rel are ground, for simplicity. The relation computed for the head is obtained as follows. (1) Let $Y_1, \ldots, Y_m$ be the set of all variables occurring in the rule head. Let r be the relation corresponding to the body of the rule. Let $\pi_{Y_1,\ldots,Y_m}(r)$ be the projection of r onto the columns corresponding to the arguments.[9] (2) Let $T_1, \ldots, T_p$ be the variables among the Y's that appear as arguments of tid in the rule head. Partition the relation $\pi_{Y_1,\ldots,Y_m}$ based on the values on columns $T_1, \ldots, T_p$. (3) For each block of the partition, compute the multiset of values in column X_i that are associated with the attribute $aggAttr_i$, and compute the aggregate agg_i of this multiset. Finally, all ground facts with the same tuple-id tid are merged into one tuple in the output.

[9]The relation corresponding to the rule body can be computed using (minor adaptations to) the functions VTOA and ATOV discussed in Ullman [Ull89a].

Semantics for the case when *db* and/or *rel* are non-ground is defined analogously.

Example 2.6.3 *Consider the relation in Figure 2 (which is a part of a database db) storing information on prices of various stocks at different exchanges (possibly in different countries) on a day to day basis, during March 1997.*

date	stock	Xge_1	...	Xge_n
01	s1	50		48
01	s2	34		40
.				
.				
02	s1	35		39
02	s2	56		43
.				

Figure 2: Stock Exchange Database

Vertical Aggregation: Our first example is the simple query

(Q_7) "For each stock, compute its average (during March 1997) closing price at the Toronto stock exchange."

This query is a conventional aggregate query expressible in conventional languages like SQL. In SchemaLog, it can be expressed as

$$toronto :: avgStockPrices[f(S) : stock{\rightarrow}S,\ avgPrice{\rightarrow}avg(P)] \longleftarrow$$
$$db :: stockInfo[stock{\rightarrow}S,\ toronto{\rightarrow}P].$$

The above rule instructs the system to retrieve the (multiset of) closing prices at the Toronto exchange for each stock, and then compute the average. Note the use of the tuple-id $f(S)$ to achieve an effect similar to SQL's "**groupby stock**". But as we shall see, grouping using tuple-id's is more powerful than SQL's **groupby**.

The query Q_7 can be extended in various ways, depending on the need and application. *E.g.*, suppose we need to compute a similar average price for stocks, but w.r.t. *every* exchange. If the number of exchanges is small and known to the user a priori, this can be expressed in the obvious way in SQL. However, SchemaLog does not require *complete* prior knowledge of the schema on the part of the user. Regardless of the number of exchanges involved, she can simply write the query

$$allXges :: avgStockPrices[f(S) : stock{\rightarrow}S,\ avgPrice(X){\rightarrow}avg(P)] \longleftarrow$$
$$db :: stockInfo[stock{\rightarrow}S,\ X{\rightarrow}P],\ X \neq stock,\ X \neq date.$$

This rule creates a database (or view) *allXges* and computes for each exchange the average price of each stock at that exchange.

Next, suppose that *stockInfo* stores information pertaining to a *whole year*. Suppose also that there is, in addition, another relation in the database – $dates2weeks(D, W)$[10] – that maps dates into week numbers. E.g., assuming the financial year starts in April and closes in March, we would expect $dates2weeks$(04-01-96, 1) and $dates2weeks$(03-31-97, 52) to hold. Now, consider the query

(Q₈) "For each stock, compute the weekly average closing prices at each of the exchanges."

This can be expressed as:

$allXges :: weeklyAvgs[f(S, W) : stock \rightarrow S,\ weekNo \rightarrow W,\ avgPrice(X) \rightarrow avg(P)] \longleftarrow$
$\quad db :: stockInfo[date \rightarrow D,\ stock \rightarrow S,\ X \rightarrow P],\ X \neq stock,\ X \neq date,\ dates2weeks(D, W).$

Horizontal Aggregation: Consider the query

(Q_9) " For each stock, compute its daily average closing price across various exchanges."

Note that unlike conventional aggregate queries which involve collecting values occurring in a column (or more) based on some grouping criterion, this involves collecting the values appearing in a row! Again, when the number of exchanges is small and known to the user a priori, one can express this query in SQL. In SchemaLog, without a detailed knowledge of the schema, the user can express Q_9 using the rule

$$xgeWiseAvg :: daily[g(S, D) : date \rightarrow D,\ stock \rightarrow S,\ avgPrice \rightarrow avg(P)] \longleftarrow$$
$$db :: stockInfo[date \rightarrow D,\ stock \rightarrow S,\ X \rightarrow P],\ X \neq stock,\ X \neq date.$$

Note that by the choice of the tuple-id $g(S, D)$, the rule instructs the system to perform a *horizontal aggregation*. This query assumes (reasonably) that stock and date uniquely determine the closing prices at each of the exchanges. In other words, stock and date form a key for the relation *stockInfo*. As another example, consider the query

(Q_{10}) "For each stock, find the daily maximum and minimum closing price over all exchanges, as well as the exchanges at which such prices prevailed."

Even assuming a complete knowledge of the schema, whenever a number of exchanges are involved (which is a typical situation), expressing this query in SQL

[10]Indeed, this may be realized as a virtual relation, implemented as an external function call, but we may assume without loss of generality that it is accessible via a programming predicate call such as $dates2weeks(D, W)$.

would involve writing a complicated program involving many temporary relations. In SchemaLog, this is accomplished elegantly.

$xgeWiseAgg :: daily[g(S, D) : date{\rightarrow}D,\ stock{\rightarrow}S,\ max{\rightarrow}max(P),\ min{\rightarrow}min(P)]$
$\quad\longleftarrow db :: stockInfo[date{\rightarrow}D,\ stock{\rightarrow}S,\ X{\rightarrow}P],\ X \neq stock,\ X \neq date.$

$xgeWiseAgg :: daily[g(S, D) : maxXge{\rightarrow}X_{max},\ minXge{\rightarrow}X_{min}] \longleftarrow$
$\quad xgeWiseAgg :: daily[g(S, D) : max{\rightarrow}P_{max},\ min{\rightarrow}P_{min}],$
$\quad db :: stockInfo[date{\rightarrow}D,\ stock{\rightarrow}S,\ X_{max}{\rightarrow}P_{max},\ X_{min}{\rightarrow}P_{min}],$
$\quad X_{max} \neq date,\ X_{max} \neq stock,\ X_{min} \neq date,\ X_{min} \neq stock.$

The first rule computes the daily maximum and minimum closing prices for each stock. The second rule derives the names of the associated exchanges by checking off the maximum and minimum prices against the various exchanges in *stockInfo*. Note that tuples of the output relation *daily* are assembled piecemeal in that different rules compute values of different attributes. The above rules assume that the daily maximum and minimum prices occur at unique exchanges. If this assumption cannot be made, it means more than one exchange could close at the maximum and/or minimum price for a given stock. In this case, the output to the query should contain a tuple for each exchange with the maximum/minimum closing price. We leave it as a simple exercise to the reader to modify the tuple-id used in the rules above to achieve this effect.

Global Aggregation: There are situations where we might need to perform aggregates on (multisets) of values retrieved from positions more general than just rows or columns. As a first example, consider

(Q_{11}) "For each stock and each week (number), compute the average closing price over all exchanges."

The output to the query must be of the form $weekly(WeekNo, Stock, Avg)$ with the obvious meaning. The problem is that the multiset of values on which the averaging must be performed for a given stock and week number, is actually contained in a "rectangular block" within the relation *stockInfo*. While it is not clear how such queries can be expressed in SQL at all, the following rule in SchemaLog expresses it in a straightforward manner.

$global :: weeklyAllXges[f(S, W) : stock{\rightarrow}S,\ week{\rightarrow}W,\ avgPrice{\rightarrow}avg(P)] \longleftarrow$
$\quad db :: stockInfo[date{\rightarrow}D,\ stock{\rightarrow}S,\ X{\rightarrow}P],\ X \neq stock,\ X \neq date,$
$\quad dates2weeks(D, W).$

To appreciate the effect of attribute names in influencing the way in which values are grouped into multisets, notice that the SchemaLog rules for queries Q_8 and Q_{11} are *almost* identical. In particular, the rule bodies are identical and the tuple-id's used in the rule head are identical. However, while Q_8 computes a series of vertical (conventional) aggregates, Q_{11} computes one global aggregate. This dramatic difference arises because in Q_8 individual multisets of prices are grouped and associated with the attribute $avgPrice(X)$, for each exchange X, before the average is computed. In Q_{11}, by contrast, all these prices are grouped into one multiset associated with the attribute $avgPrice$ (a constant), and then the (global) average is computed.

Aggregation over arbitrary collections of values (retrieved from different relations or even databases) can be quite conveniently expressed in SchemaLog, in a manner similar to that illustrated by the example of query Q_{11}.

We call such aggregation over arbitrary collections *global aggregation*. Note that in general, the global aggregation *cannot* be simulated by a sequence of horizontal and vertical aggregations. This is the case when the aggregate function is not "additive". Average is an example of a non-additive function. For instance, $avg(\{2,3,4,5,6\}) = 4 \neq avg(avg(\{2,3\}), avg(\{4,5,6\})) = 3.75$.

Our last example of this section illustrates how the concept of arity of predicates can be elegantly captured in SchemaLog.

Example 2.6.4 *Revisit query* Q_3 – "Find the names of all the binary relations in which the token 'john' appears" *– from Section 2.6.1. We now show how this query can be expressed in a succinct manner in SchemaLog. The idea is to make use of a 'system' relation defined using SchemaLog with aggregation, called* arity*. This relation would maintain information on the arity of each relation (in each database) in the federation. The following program illustrates how this relation is defined and how it is utilized for expressing query* Q_3.

$system :: arity[f(D,R) : db{\rightarrow}D,\ rel{\rightarrow}R,\ ary{\rightarrow}count(A)] \longleftarrow D :: R[A].$
$whereabouts(X,D,R) \longleftarrow D :: R[A{\rightarrow}X], system :: arity[db{\rightarrow}D,\ rel{\rightarrow}R,\ ary{\rightarrow}2].$
$?- whereabouts(john, DB, Rel)$

We close this section by noting that, using the power of higher-order variables and by a clever manipulation of tuple-id's for grouping, the user can express a rather powerful class of aggregate computations in SchemaLog. These remarks hold even when the suite of basic aggregates (avg, count, max, min, sum) available in normal implementations were to be augmented with other functions implemented via external function calls.

The dynamic restructuring and horizontal or block aggregation capabilities offered by the flexible syntax of SchemaLog indicate that SchemaLog can be used to develop a theoretical foundation for OLAP (On-Line Analytical Processing) ([CCS95]), a fledgling technology with tremendous practical potential, lacking clear foundations. Indeed, [GLS96] shows that the querying and restructuring capabilities of SchemaLog can be visualized in terms of four fundamental restructuring algebraic operators, augmented by classical algebraic operators. This paper also develops such an algebra in the context of a two-dimensional data model called the *tabular data model* and proves that it is complete for all generic, computable transformations. [GLS95] discusses in detail how the tabular data model and the tabular algebra can serve as a foundation for OLAP.

In Chapter 4, we develop a language called *SchemaSQL*, drawing on the inspiration from the SchemaLog experience. We also illustrate the usefulness of *SchemaSQL* for OLAP applications.

2.7 Comparison With Other Logics

The notion of "higher-orderness" associated with a logic is ill-defined. Chen *et. al.* [CKW90] point this out and provide a clear classification of logics based on the order of their syntax and semantics. It is generally believed that higher-order syntax would be quite useful in the context of object-oriented databases, database programming, and schema integration. In this section, we compare SchemaLog with existing higher-order logics. We also comment on the "design decisions" made in the development of SchemaLog.

HiLog: *HiLog* (Chen *et. al.* [CKW90]) is a powerful logic based on higher-order syntax but with a first-order semantics. Parameters are arityless in this language and the distinction between predicate, function, and constant symbols is eliminated. HiLog terms could be constructed from any logical symbol followed by any finite number of arguments. HiLog also blurs the distinction between the atoms and terms. Thus, the language has a powerful syntactic expressivity and finds natural applications in numerous contexts (see [CKW90] for details). HiLog has a sound and complete proof theory. [CKW89] discusses the applicability of HiLog as a database programming language. The higher-order syntactic features of the language find interesting applications for schema browsing, set operations, and as an implementation vehicle for object-oriented languages. From the viewpoint of MDBS interoperability, though HiLog has the concept of arityless-ness, the lack of a means in its syntax to refer to "places" corresponding to attributes or "method" names makes it cumbersome to express queries that range over multiple databases (or even multiple relations within the same database - see Section 2.6.1). Hence HiLog (without further extensions) seems to be unsuitable for the purpose of interoperability.

F-logic: Kifer *et. al.* [KLW95] provide a logical foundation for object-oriented databases using a logic called *F-logic*. Like HiLog, F-logic is a logic with a higher-order syntax but a first-order semantics[11]. The logic is powerful enough to capture the object-oriented notions of complex objects, classes, types, methods, and inheritance. F-logic also has a schema browsing facility which hints at the possibility of its application for interoperability. The syntax of F-logic, unlike that of SchemaLog, was not designed with interoperability as one of the main goals. Thus, using F-logic for MDBS interoperability admits several alternatives, depending on how an MDBS is modeled within F-logic syntax. In [LS95] we undertake a detailed study of the various possibilities for modeling MDBS in F-logic as well as other proposed higher-order logics and contrast these approaches with the SchemaLog based approach for interoperability. Based on our analysis, we have derived the following conclusions in respect of approaches based on F-logic. Every approach based on F-logic suffers from either or both of the following drawbacks (while all of the F-logic based approaches known to us suffer from drawback 1).

1. **"Access path" violation:** In the context of interoperability in an MDBS, it is natural to require that a relation cannot be referred to without asserting the existence of a database it belongs to, and an attribute cannot be referred to without indicating a relation it is defined on, and so forth. The syntax makes it impossible to enforce this access path at the language level.

2. **Closure property violation:** Any attempt at capturing interoperability should ensure that a full atom specifying the existence of a database having a certain value for given relation, attribute, and tid, needs to imply an expression that asserts the existence

[11]When non-monotonic method inheritance is not considered.

of the database, the relation, etc. In SchemaLog, this notion is naturally captured in the model theory. Many of the approaches based on F-logic do not enforce this property within the logic itself, making it necessary to write programs to enforce such constraints.

Though SchemaLog uses some concepts and some techniques similar to those used for HiLog and F-logic, it has some important technical differences which include the following:

(1) Function symbols in SchemaLog are interpreted extensionally, whereas in HiLog, they are interpreted intensionally. This feature allows the classical techniques for Skolemization (and hence proof procedure) to be used for SchemaLog (with minor modifications to account for its syntax and the notion of a closed structure).

(2) SchemaLog features position independence (achieved by using attribute names and tuple id's). Position independence allows us to ignore the argument positions of relations in a database; they can be referred to unambiguously through their names. HiLog is position dependent. While F-logic is position independent (it has names for its methods/attributes), the way the SchemaLog semantic structure interprets the attribute names is significantly different from the way the F-logic structure interprets its method names. This is true even if one strips off (i) those aspects of an F-logic structure which are needed only for those methods which take arguments (unlike the attributes of a relation) and (ii) the aspects needed mainly for inheritance.

HOL: A higher-order language for computing with labeled sets is introduced in Manchanda [Man89]. The language supports structured data, object-identity, and sets. This also belongs to the above class of languages in that its semantics is first-order. This paper also illustrates a template mechanism to define the database schema. But it is not obvious how to extend this language to a framework which would support queries over higher-order objects across multiple databases.

COL: Abiteboul and Grumbach [AG87] introduce a logic called COL for defining and manipulating complex objects. COL achieves the functionality for manipulating complex objects by introducing what are called (base and derived) "data functions". The syntax as well as the semantics of COL is higher-order. The syntax does not support the constructs necessary for interoperability.

Approach based on Annotated logic: In recent work, Subrahmanian [Sub94] has studied the problem of integrating multiple deductive databases featuring inconsistencies, uncertainties, and non-monotonic forms of negation. He proposes an approach based on annotated logics ([Sub87], [KL88], [KS92]) for realizing a "mediator" between the component knowledge-bases. We observe that the contribution of this paper neatly complements that of SchemaLog for data integration, in that SchemaLog helps resolve conflicts arising from data/meta-data interplay whereas Subrahmanian's framework allows to handle inconsistencies between (the data in) component databases. We can easily augment the framework of SchemaLog either with annotations (in the spirit of annotated logics) or with the *Information Source Tracking* framework proposed by Sadri [Sad91] and studied by Lakshmanan and Sadri [LS94]. The resulting language will be powerful enough to handle both kinds of inconsistencies.

The SchemaLog Approach: In principle, one could augment HiLog or F-logic with the facilities for naming individual schemas as well as naming attributes (in the case of HiLog). In our project, we have chosen to start from a "neutral zone" and try to build a logic that is as simple as possible while effectively solving the problem on hand. One of the benefits of this approach has been with regard to ease of implementation (see Section 2.8). The development of a relational calculus inspired by SchemaLog syntax and of an algebra with an equivalent expressive power (See Chapter 3) has had a strong impact on the ease and efficiency of our implementation of SchemaLog. In [Law93] it has been pointed out based on implementation experience that there are many difficulties in implementing F-logic with its complex semantics and proof-theory. Indeed, this has led some researchers to investigate implementations of languages based on restricted versions of F-logic ([Dob95]). We are not aware of an algebra corresponding to (even restricted versions of) F-logic. Secondly, for extending SchemaLog to cater for an OO data model, there is really no need to incorporate all the features of OODBs within the logic: we simply need a construct which will act as an "interface" to an OODB and retrieve information from it. The details of how the rich features of an OODB are modeled can be left outside the language for so far as the purpose of interoperability goes. We also remark that making SchemaLog arityless (like HiLog) (also see discussions on molecular programs in Section 2.5) presents no problems for the semantics. In our work, we have chosen to keep the logic no more complex than necessary for the problem studied here. We remark that even with this simplicity, SchemaLog appears to be quite powerful and easy to program in, for several applications (see Section 2.6).

2.8 Implementation

In this section, we briefly discuss our implementation of the querying fragment of SchemaLog on an MDBS consisting of schematically disparate INGRES databases. In principle, we can use the equivalence to predicate calculus result of Section 2.4 to realize an implementation on Prolog. But, such an implementation would clearly be inefficient - the existing federation would need to be rewritten to a first-order reduced form - an expensive process in itself. Instead, we adopt the following approach.

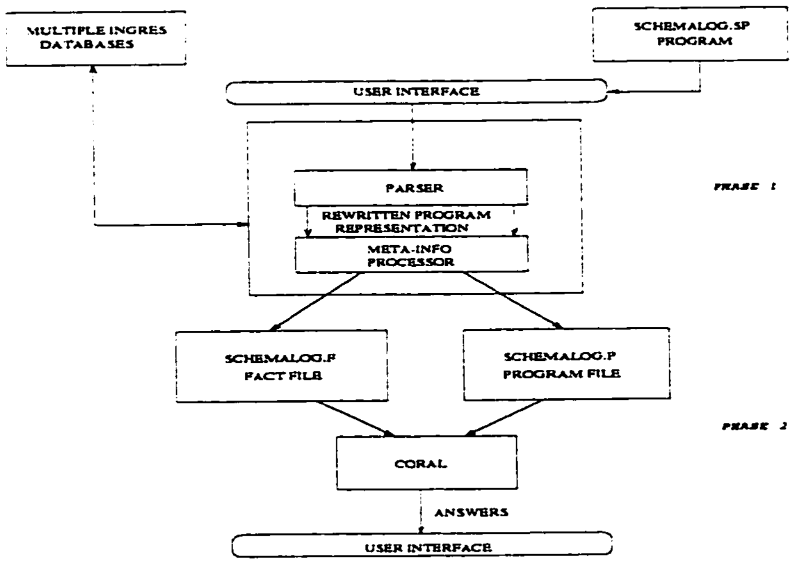
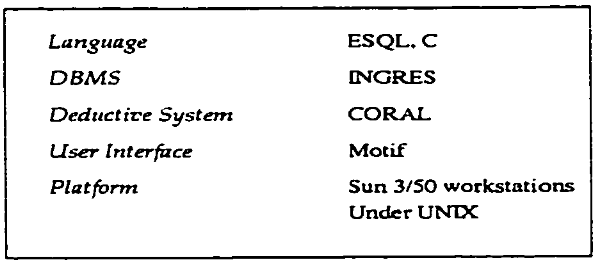


Figure 3: SchemaLog System Architecture

Two important aspects of SchemaLog are *(i)* its higher-order features to access schema information from multiple databases, and *(ii)* deduction. A significant feature of our implementation is that these two aspects are handled independently. The schema information is manipulated using operators of the extended algebra, $\mathcal{SQA}$ (to be discussed in Chapter 3) implemented using INGRES Embedded-SQL (ESQL). The deductive DBMS CORAL [RSS92] is used for recursive query processing. Figure 3 shows the architecture of our implementation. Since user programs can involve complex interactions between schema manipulation and deduction, our implementation integrates these two functionalities from ESQL and CORAL closely.

Phase 1 of our implementation is concerned with extracting the schema related information of databases in the federation and converting it to a "first-order" form. This phase essentially makes use of the extended algebra ($\mathcal{SQA}$) operators. The implementation compiles the SchemaLog program into an algebraic form. During this process, various optimization strategies suggested by the properties of the algebraic operators ([LSS95]) are employed to minimize the cost of fetching the meta-information as well as to reduce the amount of information that needs to be processed in Phase 2. In the second phase, the inference engine of CORAL and its rich suite of recursive query optimization strategies are exploited for efficient query processing. The system sports a pleasant user interface capable, among other things, of a schema browsing facility. Further details of this implementation are discussed in [LSPS95. Pap94]. Figure 4 shows our implementation platform for SchemaLog.



Language	ESQL, C
DBMS	INGRES
Deductive System	CORAL
User Interface	Motif
Platform	Sun 3/50 workstations Under UNIX

Figure 4: Implementation Particulars of SchemaLog

As demonstrated in this implementation, the simplicity of SchemaLog has resulted in an elegant design, and in its easy realization *even within the framework of current relational database systems.* Our ongoing work involves using the database storage manager EXODUS [CDRS86] for storing the output of Phase 1. We expect this to yield a significant gain in performance, as CORAL has a direct interface to EXODUS for storing and manipulating persistent relations. Our ongoing implementation includes the full power of SchemaLog programming language (allowing SchemaLog molecules, as opposed to just programming predicates, in rule heads).

2.9 Conclusions

Developing a declarative approach to integrating information from multiple heterogeneous data sources is a major goal of our research. We have taken the first step toward this goal in this chapter, by developing a simple logic called *SchemaLog* which is syntactically higher-order but has a first-order semantics. SchemaLog provides for interoperability among a federation of multiple relational databases. We developed a fixpoint theory for the definite clause fragment of SchemaLog and showed its equivalence to the model-theoretic semantics. We also developed a sound and complete proof procedure for *all* clausal theories. We established the correspondence between SchemaLog and first-order predicate calculus and provided a reduction of SchemaLog to predicate calculus. We illustrated the simplicity and power of SchemaLog with a variety of applications involving database programming (with schema browsing), schema integration, schema evolution, cooperative query answering, and aggregation. We also highlighted our implementation of SchemaLog realized on a federation of INGRES databases.

In view of the reduction to predicate calculus (see Section 2.4), one may ask the question, why not use standard predicate calculus for the applications envisaged here. The following are some of the reasons why our approach would be superior to one based on first-order reduction. (1) As we have demonstrated, programming in SchemaLog is more natural and much more concise. (2) As proved in Section 2.4, it is impossible to use classical predicate calculus for interoperability in a schema preserving manner. (3) The notion of closure (Section 2.5.1) is directly captured in the SchemaLog unification theory. In a first-order encoding based approach, closure needs to be captured in a roundabout way by adding axioms of the form '$call_{i-1}(\cdots) \leftarrow call_i(\cdots)$', $i = 2, 3, 4$, to the reduced program. Clearly, this leads to inefficiency in query evaluation. (4) SchemaLog is much better equipped with the wherewithal for developing a paradigm capable of addressing the interoperability issues arising in MDBS featuring multiple data models.

We note that though in this chapter we have confined ourselves to interoperability among multiple relational databases, the contributions here lay the foundations for many of the later chapters in this thesis. In particular, in Chapter 5, we extend SchemaLog to provide for interoperability among MDBS featuring disparate data models such as ER and Network models as well as information sources on the Web. SchemaSQL, discussed in Chapter 4 also draws its inspiration from SchemaLog.

Chapter 3

Algebra and Calculus

In this chapter, we develop an algebra by extending the conventional relational algebra with some new operations so that the resulting algebra is capable of accessing the database names, relation names, and attribute names besides the values, in a federation of databases, in a uniform manner. We also develop a calculus (along the lines of classical relational calculus) inspired by a fragment of SchemaLog that is useful for federation querying. An important contribution of this chapter is the study of various notions of *safety* associated with the calculus language. We first introduce a fragment of the algebra that is appropriate for *querying* data and meta-data in a uniform manner, and extend the equivalence result between classical relational algebra and relational calculus to a framework which manipulates data and schema uniformly. Since SchemaLog is capable of performing federation *restructuring* as well, we introduce algebraic operators for restructuring information to conform to different schemas.

Study of such an algebra is important in its own right. A SchemaLog query compiled into an abstract algebraic form would hide the low level algorithmic details of its implementation. It would better reveal the various query optimization opportunities suggested by the properties of the algebraic operations. Thus, such a study is fundamental to the development of strategies for efficiently realizing a SchemaLog based database programming platform in a federation of existing database management systems.

3.1 Overview of the Chapter

In Section 3.2, we discuss the important requirements we seek in our algebra. Based on these requirements, we develop the schema (as well as data) querying algebra in Section 3.3. In Section 3.4, we define a fragment of SchemaLog capable of federation querying and prove that the expressive power of the algebra is no less than that of the query language. In Section 3.5, we develop a calculus language inspired by SchemaLog and study various notions of safety that naturally arise in the context of this language. We also study the relative expressive powers of the calculus, algebra, and querying languages and establish that they have an equivalent expressive power. In Section 3.6, we develop the algebra capable of restructuring information to conform to different schemas. Finally in Section 3.7, we make some important observations in connection with the many issues studied in this chapter.

3.2 Requirements for the Algebra

In this section, we discuss some of the important requirements that dictate the development of our algebra.

1. The algebra must be sufficiently expressive in the sense that it should support the possibility of compiling query and restructuring programs written in SchemaLog into algebraic expressions which when evaluated iteratively will return a result that is equivalent to the result expected from the SchemaLog program.
2. The operations must be as simple as possible and should admit efficient implementation.

We first observe that SchemaLog supports two kinds of predicates: (i) *database predicates*, and (ii) *programming predicates*. Database predicates correspond to databases whose schema information (*i.e.* the names of the databases, names of the relations and the attributes in the databases) is of interest and hence is given first class status. This is the case for relations stored in an existing database (the so-called *edb*) as well as relations constructed by database programs written in SchemaLog, where the associated schema information is regarded just as important as the data. Programming predicates are often used as a device for temporary storage of intermediate computations by programs, and sometimes for holding answers to queries. In this case, no attention is paid to the schema of such relations. Recall Example 2.6.2 of Chapter 2 for an instance of a SchemaLog program in which both database predicates and programming predicates are used in an essential way. It should be pointed out that in classical deductive database query languages such as Datalog, the only kind of relations supported are programming relations.

From the preceding discussion, it is clear that our algebra must manipulate two kinds of objects – database relations and programming relations. Note that the interface provided by classical relational algebra to the relations in a database essentially treats them as "programming" relations. This is because it is impossible to query or manipulate meta-data using this interface. Our extended algebra has three kinds of operators.

1. Classical Relational Algebra (RA) operators: these are capable of querying (the data in) database as well as programming relations.

2. Operators which query database relations (both data and schema) and present the output in the form of programming relations. Thus, they map relations in a database to programming relations.

3. Operators which take as input programming relations (and some parameters) and structure the information in them in specified ways. Thus, such operators map programming relations into database relations.

Operators of type (2) (Section 3.3) and type (3) (Section 3.6) are new and are unique to our algebra. Before presenting the definitions of the operators, we remark that in classical RA, one can refer to attributes (which are schema components) either by position or by name. However, this does **not** mean that their schema is given a first class status. The point is that schema information, when it is not explicitly represented in a relational form, cannot be retrieved or restructured using classical RA. Our algebra facilitates powerful meta-data querying and restructuring, besides providing for conventional data querying. We first introduce the type (2) operators.

3.3 Schema Querying Algebra

In this section, we introduce the querying operators in our algebra. The definitions of classical relational algebra — *selection* (σ), *projection* (π), *cartesian product* ($\times$), *union* ($\cup$), *difference* ($-$), — are as usual: no modifications are necessary. As remarked in the opening paragraph of this chapter, we study in depth, the various notions of safety arising in a higher-order calculus language. With an aim towards studying the relative expressive powers of the various safe fragments of the calculus language and the algebra introduced in this section, whenever we define a new operator, we also define an operator that is its variant, and describe the scenario in which the variant operator is applicable.

Definition 3.3.1 (Fetching database names) *The first new operator we introduce, δ, is 0-ary and returns the set of names of all the databases in the federation of databases.*

$\delta() = \{d \mid d \textit{ is the name of a database in the federation }\}$

For example, $\delta()$ against the university federation of Example 1.1.1, would return the unary relation: $\{univ\_A, univ\_B, univ\_C\}$.

The δ operator is useful in scenarios in which the names of the databases in a federation are not known apriori. To cater to the scenario in which the names of the databases of interest should be restricted to a set of pre-determined names, we introduce the following variant of the δ operator.

Definition 3.3.2 *The operator $\hat{\delta}$ is a unary operator that takes a unary relation (i.e. a set) as input and returns those entries in the input relation that correspond to the names of databases in the federation.*

$\hat{\delta}(s) = \{d \mid d \in s \ \wedge \ d \textit{ is the name of a database in the federation }\}$

For example, $\hat{\delta}(\{univ\_A, univ\_D, univ\_E\})$ against the university federation would return the relation $\{univ\_A\}$.
The second operator is capable of querying relation names in databases.

Definition 3.3.3 (Fetching relation names) *The relation querying operator is a unary operator that takes a unary relation as input and returns a binary relation, as follows.*

$\rho(p) = \{\langle d, r\rangle \mid d \in p.$ *d is a name of a database in the federation, r is a relation name in d*$\}$

For each database name d in the input set p, ρ associates d with the name of each relation that is part of the database d in the federation.

For example, let relation $p = \{univ\_A, univ\_C\}$. $\rho(p)$ against the university federation would yield the relation: $\{\langle univ\_A, pay\_info\rangle, \langle univ\_C, cs\rangle, \langle univ\_C, ece\rangle, \langle univ\_C, math\rangle\}$.

From the definition of ρ above, it might appear that $\widehat{\delta}$ can be simulated using ρ and π. However, note that this is not the case - the counter example is a scenario in which there exists a database that does not contain any relations. Similar observation applies for the other variant operators (introduced further down in this section) as well.
Similar to the $\widehat{\delta}$ operator, we now introduce a variant of the ρ operator.

Definition 3.3.4 *Operator $\widehat{\rho}$ takes a binary relation as input and returns a subset of the input relation as shown below.*
$\widehat{\rho}(t) = \{\langle d, r\rangle \mid \langle d, r\rangle \in t,$ *d is a name of a database in the federation, r is a relation name in d*$\}$

The next operator in our algebra is intended to extract attribute names from relations of the federation.

Definition 3.3.5 (Fetching attribute names) *Operator α takes a binary relation as argument and returns a ternary relation.*

$\alpha(q) = \{\langle d, r, a\rangle \mid \langle d, r\rangle \in q,$ *d is a database in the federation, r is a relation name in d, and a is an attribute name in the scheme of r*$\}$.

For each $\langle d, r\rangle$ pair appearing in q such that r is a relation in federation database d, α associates to the pair, names of each attribute in the scheme of r.

For example, let $q = \{\langle univ\_C, cs\rangle\}$. In the context of the university federation, $\alpha(q)$ would return the relation: $\{\langle univ\_C, cs, category\rangle, \langle univ\_C, cs, avg\_sal\rangle\}$.

We now present operator $\widehat{\alpha}$, variant of α.

Definition 3.3.6 *This operator takes a ternary relation as input and returns its subset as defined below.*
$\tilde{\alpha}(u) = \{\langle d, r, a\rangle \mid \langle d, r, a\rangle \in u$, *d is a database in the federation, r is a relation name in d, and a is an attribute name in the scheme of r*$\}$.

Before we formaliy present the last new operator γ of our querying algebra, some basic definitions are in order.

Definition 3.3.7 *A* pattern *is a sequence* $\langle p_1, \ldots, p_k\rangle$, $k \geq 0$, *where each* p_i *is of one of the forms:* '$a_i \rightarrow v_i$', '$a_i \rightarrow$ ', ' $\rightarrow v_i$', ' $\rightarrow$ '. *Here* a_i *is called the* attribute component *and* v_i *is called the* value component, *of* p_i. *Let r be any relation.*
'$a_i \rightarrow v_i$' *is* satisfied *by a tuple t in relation r if* $t[a_i] = v_i$; *in this case, attribute* a_i *is called the* witness of satisfaction.
'$a_i \rightarrow$ ' *is* satisfied *by a tuple t in relation r if* a_i *is an attribute in r;* a_i *is the witness of satisfaction.*
' $\rightarrow v_i$' *is* satisfied *by a tuple t in relation r if* $\exists$ *an attribute* a_i *in the scheme of r such that* $t[a_i] = v_i$; *attribute* a_i *is the witness of satisfaction.*
' $\rightarrow$ ' *is* satisfied *by a tuple t in relation r if* a_i *is an attribute in the scheme of r such that* $t[a_i] = v_i$; a_i *is the witness of satisfaction.* ' $\rightarrow$ ' *is trivially satisfied by every tuple t in relation r.*

A pattern $\langle p_1, \ldots, p_k\rangle$ is satisfied by a tuple t in relation r with witness of satisfaction $\{a_1, \ldots, a_k\}$ provided $\langle p_i\rangle$ is satisfied by tuple t with witness a_i, $i = 1, \ldots, k$.

Operator γ allows us to relate data to meta-data. It takes a binary relation as input, and a pattern as a parameter and returns a relation that consists of tuples corresponding to those parts of the database where the queried pattern exists. Formally,

Definition 3.3.8 *Let s be a binary relation and* $\langle p_1, \ldots, p_k\rangle$ *be a pattern as defined in Definition 3.3.7. Then,*
$\gamma_{\langle p_1, \ldots, p_k\rangle}(s) = \{d, r, a_1, v_1, \ldots, a_k, v_k \mid \langle d, r\rangle \in s \wedge$ *d is a database in the federation* $\wedge$ *r is a relation in d* $\wedge \exists$ *a tuple* $t \in r$ *such that t satisfies* $\langle p_1, \ldots, p_k\rangle$ *with witness* $\{a_1, \ldots, a_k\} \wedge t[a_1, \ldots, a_k] = (v_1, \ldots, v_k)\}$.

Note that when the pattern is empty, $\gamma_{\langle\rangle}(s)$ would return the set of all pairs $\langle d, r\rangle \in s$ such that r is a *non-empty* relation in the database d in the federation.

Example 3.3.1 *The operation* $\gamma_{\langle\ \rightarrow\text{'secretary'},\ \rightarrow\ \rangle}(s)$ *against the university databases of Example 1.1.1 will yield the relation in Figure 5.*

univ_A	pay_info
univ_B	pay_info
univ_C	cs
univ_C	ece

univ_A	pay_info	category	secretary	dept	cs
univ_A	pay_info	category	secretary	category	secretary
univ_A	pay_info	category	secretary	avg-sal	35K
univ_C	ece	category	secretary	category	secretary
univ_C	ece	category	secretary	avg-sal	30K

Figure 5: (a) Relation s and (b) output of $\gamma_{\langle\ \rightarrow\text{'secretary'},\ \rightarrow\ \rangle}(s)$

Finally, operator γ also has its variant, which we define below.

Definition 3.3.9 *Let u be a ternary relation and $\langle p_1, \ldots, p_k\rangle$ be a pattern. Then,* $\hat{\gamma}_{\langle p_1,\ldots,p_k\rangle}(u) = \{d, r, a_1, v_1, \ldots, a_k, v_k \mid \langle d, r, a_1\rangle, \ldots, \langle d, r, a_k\rangle \in u \wedge d$ *is a database in the federation* $\wedge$ r *is a relation in* d $\wedge$ $\exists$ *a tuple* $t \in r$ *such that* t *satisfies* $\langle p_1, \ldots, p_k\rangle$ *with witness* $\{a_1, \ldots, a_k\} \wedge t[a_1, \ldots, a_k] = (v_1, \ldots, v_k)\}$.

We remark that operators σ, π, $\times$, $\cup$, $-$, and each of the main schema querying operators (or its variant) of our extended algebra form an independent set of operators: each operator *cannot* be simulated using one or more of the other operators. In particular, note that given a binary relation q, the effect of operations $\alpha(q)$ and $\pi_{1,2,3}(\gamma_{\langle\ \rightarrow\ \rangle}(q))$ is *not* the same: $\alpha(q)$ contains in its output, tuples of the form $\langle d, r, a\rangle$ such that $\langle d, r\rangle \in q$, r is *any* relation (possibly empty) in the database d, and a is an attribute in r's scheme. On the other hand, the output of $\pi_{1,2,3}(\gamma_{\langle\ \rightarrow\ \rangle}(q))$ includes only non-empty relations.

However, note that each of the main schema querying operators can simulate its variant operator. As an example, $\hat{\delta}(s) \equiv \delta() \cap s$. The variant operators are needed for scenarios in which the main operator would be too costly to be realized. Also, as can be seen later in this chapter, the various notions of safety are intrinsically related to the fragments of the algebra obtained using the variant operators.

Example 3.3.2 Query Q_2 of Section 2.2, "*List similar departments in univ_B and univ_C that have the same average salary for similar categories of staff*" can be expressed in our extended algebra as:

$$\pi_{\$4,\$5,\$6}\ \sigma_{\$4=\$10\ \wedge\ \$5=\$8\ \wedge\ \$6=\$12}\ (\sigma_{\$5\neq\text{'}category\text{'}}(\ \gamma_{\langle category\ \rightarrow\ ,\ \rightarrow\ \rangle}$$
$$(\{\langle univ\_B, pay\_info\rangle\})\ )\ \times\ \gamma_{\langle category\ \rightarrow\ ,\ avg\_sal\ \rightarrow\ \rangle}(\ \rho(\{\langle univ\_C\rangle\})\ ))$$

Based on the operators we have defined so far, we now define various fragments of the querying algebra. The motivation for such a fragmentation is the following. Consider the implementation of SchemaLog atop a huge federation spanning multiple databases across a wide geographic layout. Clearly, for practical reasons, it would be infeasible to process queries that require an access of the entire federation. However, this is not the case with queries that restrict attention to parts of the federation and hence our implementation should admit such queries. The algebra we present below, is designed with such a need in mind. Each fragment of the algebra, successively restricts attention to specific databases in the federation, specific relations in specific databases and so on.

Definition 3.3.10 *$\mathcal{SQA}$, d-$\mathcal{SQA}$, r-$\mathcal{SQA}$, a-$\mathcal{SQA}$*
We will denote the algebra consisting of the classical algebra operations as well as the querying operators, δ, ρ, α, and γ as $\mathcal{SQA}$. Based on the variant operators, we also define the following fragments of $\mathcal{SQA}$.

db-bounded $\mathcal{SQA}$ *(d-$\mathcal{SQA}$):*	*The algebra obtained from $\mathcal{SQA}$ by substituting the δ operator with operator $\hat{\delta}$.*
rel-bounded $\mathcal{SQA}$ *(r-$\mathcal{SQA}$):*	*The algebra obtained from d-$\mathcal{SQA}$ by substituting the ρ operator with operator $\hat{\rho}$.*
attr-bounded $\mathcal{SQA}$ *(a-$\mathcal{SQA}$):*	*The algebra obtained from r-$\mathcal{SQA}$ by substituting α and γ with operators $\hat{\alpha}$ and $\hat{\gamma}$ respectively.*

It is evident from the above definition that *d-$\mathcal{SQA}$* is incapable of listing the names of all the databases in a federation. Intuitively, this algebra is suited for a scenario in which the names of the databases of interest should be restricted to names known apriori (or obtained using the other operators in the algebra). A similar comment (suitably modified with relation names and attribute names) applies for the other fragments of $\mathcal{SQA}$ as well: *r-$\mathcal{SQA}$* restricts attention to specific relations in specific databases and *a-$\mathcal{SQA}$* is applicable in scenarios in which (schema level) operations should be restricted to specific columns of specific relations in specific databases.

3.4 SchemaLog Based Query Language

In general. SchemaLog permits not only querying (both data and schema) of component databases, but also *restructuring*. For instance, as shown in Section 2.6.2, it is straightforward to restructure the information in database *univ_B* of Example 1.1.1 to conform to the schema of the database *univ_A*, using a simple SchemaLog program. In this section, we restrict attention to the part of SchemaLog that performs (schema and data) *querying*, the fragment to be precisely defined later in this section.

Motivated by the same needs for which in the previous section we defined the various fragments of the algebra, we also define the progressively restrictive fragments of the SchemaLog-based query language. Thereto, we define the following notions of *limited*, *db-limited*, *rel-limited*, and *attr-limited* variables appearing in a SchemaLog rule.

Definition 3.4.1 *A variable appearing in a SchemaLog rule is* limited *if it appears as an argument of a non-negated subgoal or is equated to a constant or a limited variable (perhaps through a chain of equalities). A SchemaLog rule is* safe *if all variables appearing in it are limited.*

Definition 3.4.2 *A variable appearing in a SchemaLog rule is* db-limited (rel-limited, attr-limited) *if* (a) *it appears in a non-db (non-db and non-rel, value) position of a non-negated SchemaLog subgoal having a constant or a db-limited (rel-limited, attr-limited) variable in the db (db and rel, non-value) position(s), or* (b) *it is equated to a constant or a db-limited variable (perhaps through a chain of equalities). A SchemaLog rule is* db-safe (rel-safe, attr-safe) *if all variables appearing in it are db-limited (rel-limited, attr-limited).*

Definition 3.4.3 *The* Querying Fragment of SchemaLog *($\mathcal{L}_Q$), is obtained by imposing the following constraints on the definite clause fragment of SchemaLog: –* (i) *no function symbols are allowed,* (ii) *rule heads are required to be programming predicates,* (iii) *rules are non-recursive and safe, and* (iv) *tuple-id's (used only in rule bodies) are unshared existential variables. The* db-safe (rel-safe, attr-safe) querying fragment *of SchemaLog $\mathcal{L}_Q^d$ ($\mathcal{L}_Q^r$, $\mathcal{L}_Q^a$), is obtained by imposing an additional constraint that* (v) *the rules are db-safe (rel-safe, attr-safe).*

The rationale for the above restrictions is as follows. The restriction of rule heads to programming predicates ensures that the resulting language only permits querying, as opposed to database restructuring. The restriction on tuple-id's ensures that the querying cannot depend on the internal details of tuple-id's somewhat akin to conventional relational query languages. At the same time, owing to the higher-order nature of this language, it still permits schema browsing and queries that can explore the rich semantics of schema. The restriction to the non-recursive fragment allows us to relate this language to the extended relational algebra defined earlier.

Programming in the above fragment would be based on molecules, and terms would either be constant or variable symbols. Also, programs in this language can essentially ignore the tuple-id's. The resulting database programming language is quite in line with the relational model in that, the latter also does not allow manipulation of tuple-id's. The following lemma establishes the sufficiency of $\mathcal{SQA}$ to implement safe $\mathcal{L}_Q$ programs.

Lemma 3.4.1 *Let $\mathcal{D}$ be a federation of databases (*edb*), $\mathcal{P}$ be a set of safe rules in $\mathcal{L}_Q$, and p be any predicate defined by $\mathcal{P}$. Let $\mathcal{P}(\mathcal{D})$ denote the output computed by $\mathcal{P}$ on input $\mathcal{D}$ and let $p^{\mathcal{P}(\mathcal{D})}$ be the relation corresponding to p in $\mathcal{P}(\mathcal{D})$. Then there exists an expression E, depending only on $\mathcal{P}$, in $\mathcal{SQA}$ such that $E(\mathcal{D}) = p^{\mathcal{P}(\mathcal{D})}$.*

PROOF. There are two major parts to this proof. In the first part we need to prove that each predicate defined by $\mathcal{P}$ has an equivalent expression in $\mathcal{SQA}$. The second part deals with proving that DOM, the set of all symbols appearing in $\mathcal{P}$ and in the edb relations, can be generated using $\mathcal{SQA}$.

Part I: Proof of this part is similar to [Ull89b]. Subgoals in rules in $\mathcal{P}$ consist of conventional (programming) predicates as well as SchemaLog molecules. For each subgoal S_i, let Q_i be the corresponding $\mathcal{SQA}$ expression, and let the schema of the relation corresponding to Q_i be the variables appearing in S_i. Subgoals that are programming predicates are handled as in [Ull89b]. We show how relations corresponding to subgoals that are SchemaLog molecules can be derived using $\mathcal{SQA}$. There are four cases to consider, depending on the depth.

When a subgoal S_i is a SchemaLog molecule of depth:

(1) Let S_i be X. Then $Q_i = \delta()$. If S_i is a constant d, then Q_i is simply $\{d\}$.

(2) Let S_i be $D :: R$. Then $Q_i = \rho(\delta())$. If one or more of D, R are constants, or if D and R are the same variable, then simply modify Q_i by imposing appropriate additional selection(s).

(3) If S_i is $D :: R[A_1, \ldots, A_n]$, then Q_i is essentially the expression $\alpha(\rho(\delta()))$ θ-joined with itself n-times, where θ is '$\$1 = \$1 \wedge \$2 = \$2$'. If some of the terms in S_i are constants or repeating variables, we can impose appropriate selections in Q_i.

(4) If S_i is of the form $D :: R[A_1 \rightarrow V_1, A_2 \rightarrow V_2, \ldots, A_n \rightarrow V_n]$[1], then Q_i is $\pi_{outputArgs}(\sigma_{conditions}\gamma_{\langle p_1 \ldots p_n \rangle}(\rho(\delta())))$, where p_i is an attribute/value pair of one of the forms '$\rightarrow$', '$a_i \rightarrow$', '$\rightarrow v_i$', '$a_i \rightarrow v_i$', depending on whether and where the pair $A_i \rightarrow V_i$ contains constants. $\sigma_{conditions}$ corresponds to selection conditions capturing the occurrence of constants and repeating variables in S_i, and *outputArgs* is the list of arguments corresponding to distinct variables occurring in S_i.

Now, the technique of [Ull89b] can be applied to obtain an expression for $\mathcal{P}$.

Part II: Evaluating negated subgoals involves generating complementary relations ([Ull89b]). We need to prove that $\mathcal{SQA}$ can generate DOM, the set of all constants appearing in $\mathcal{P}$ and in the databases in the federation. As our framework treats attribute names and relation names as first class citizens, the $\mathcal{SQA}$ expression generating DOM should include them in the domain. If C is the set of all constants appearing in $\mathcal{P}$, DOM is expressed the following way.

$$DOM = C \cup \delta() \cup \pi_2(\rho(\delta())) \cup \pi_3(\alpha(\rho(\delta()))) \cup \pi_4(\gamma_{\langle \rightarrow \rangle}(\rho(\delta())))$$

With these modifications, the proof is easily obtained along the lines of [Ull89b].

□

We now turn our attention to the relative expressive powers of the other fragments of $\mathcal{L}_Q$ and the algebra. The following lemma establishes that the expressive power of $\mathcal{L}_Q^d$ is no more than that of d-$\mathcal{SQA}$.

Lemma 3.4.2 *Let $\mathcal{D}$ be a federation of databases, $\mathcal{P}$ be a set of db-safe rules in $\mathcal{L}_Q^d$, and p be any predicate defined by $\mathcal{P}$. Let $\mathcal{P}(\mathcal{D})$ denote the output computed by $\mathcal{P}$ on input $\mathcal{D}$ and let $p^{\mathcal{P}(\mathcal{D})}$ be the relation corresponding to p in $\mathcal{P}(\mathcal{D})$. Then there exists an expression E, depending only on $\mathcal{P}$, in d-$\mathcal{SQA}$ such that $E(\mathcal{D}) = p^{\mathcal{P}(\mathcal{D})}$.*

[1] As discussed in Section 3.4, the tuple-id component can be ignored in $\mathcal{L}_Q$.

PROOF. The proof is similar to the proof of Lemma 3.4.1, and we present the modifications required to both parts of the above to proof.

<u>Part I:</u> The major modification from the proof of Lemma 3.4.1 is the case when a variable appears in the db-position of an atom α. Obtaining the *d-$\mathcal{SQA}$* expression when the db-position of α contains a constant is straight-forward – these constants would be the elements of the argument relation s of $\hat{\delta}$. We illustrate the proof for the scenario in which the db term is a variable for the case when the depth of α is 1. Proofs for the other cases are similar.

Let α be X. It follows from the definition of a db-safe rule of $\mathcal{L}_Q^d$ that X already occurs as (or is equated to) a db-safe variable in another atom β of the rule. Let this be the k-th variable in β and Q_j be the *d-$\mathcal{SQA}$* expression corresponding to β. Then, the *d-$\mathcal{SQA}$* expression corresponding to α, $Q_i = \hat{\delta}(\pi_k Q_j)$.

With this modification, the proof is obtained along the lines of the proof of Lemma 3.4.1.

<u>Part II:</u> The proof of this part is based on an important observation that, when we consider the $\mathcal{L}_Q^d$ fragment, we only need to restrict attention to a subset of DOM (the set of all constants appearing in $\mathcal{P}$ and in the databases in the federation). This is because, by definition, programs in the $\mathcal{L}_Q^d$ fragment can only access databases that are *named* either directly (by means of constants) or (by 'traversing' the subgoals in the program) via other named objects in the federation. Thus, given a $\mathcal{L}_Q^d$ program $\mathcal{P}$, containing k SchemaLog atoms in it, DOM_{db} of $\mathcal{P}$ is defined as:

$$DOM_{db}^0(\mathcal{P}) = DOM_{rel}^0(\phi) = DOM_{attr}^0(\phi) = C$$
$$DOM_{db}^{i+1}(\mathcal{P}) = \{s \mid \exists d \in DOM_{db}^i, s \text{ is a relation name, attribute name, or value in database } d\}$$
$$DOM_{db}(\mathcal{P}) = \cup_{0 \le i \le k} DOM_{db}^i(\phi)$$

We now prove that DOM_{db} can be generated using *d-$\mathcal{SQA}$* by presenting the corresponding *d-$\mathcal{SQA}$* expression.

$$DOM_{db}^0(\mathcal{P}) = C.$$
$$DOM_{db}^{i+1}(\mathcal{P}) = \hat{\delta}(DOM_{db}^i(\mathcal{P})) \cup \pi_2(\rho(\hat{\delta}(DOM_{db}^i(\mathcal{P})))) \cup \pi_3(\alpha(\rho(\hat{\delta}(DOM_{db}^i(\mathcal{P}))))) \cup \pi_4(\gamma_{\langle \ \rightarrow \ \rangle}(\rho(\hat{\delta}(DOM_{db}^i(\mathcal{P})))))$$

$DOM_{db}(\mathcal{P}) = \bigcup_{0 \leq i \leq k} DOM_{db}^{i}(\mathcal{P})$,

where k is the number of $\mathcal{L}_Q^d$ atoms appearing in $\mathcal{P}$.

With these modifications, the proof is obtained along the lines of the proof of Lemma 3.4.1. □

A similar expressibility result extends to the other fragments of $\mathcal{L}_Q$ and $\mathcal{SQA}$ as well. We conclude this section, by presenting these results; the proofs are very similar to that of Lemma 3.4.2 (appropriately extended to the relation and attribute fragments) and are hence omitted.

Lemma 3.4.3 *Let $\mathcal{D}$ be a federation of databases, $\mathcal{P}$ be a set of (rel-safe) rules in $\mathcal{L}_Q^r$, and p be any predicate defined by $\mathcal{P}$. Let $\mathcal{P}(\mathcal{D})$ denote the output computed by $\mathcal{P}$ on input $\mathcal{D}$ and let $p^{\mathcal{P}(\mathcal{D})}$ be the relation corresponding to p in $\mathcal{P}(\mathcal{D})$. Then there exists an expression E, depending only on $\mathcal{P}$, in r-$\mathcal{SQA}$ such that $E(\mathcal{D}) = p^{\mathcal{P}(\mathcal{D})}$.*

Lemma 3.4.4 *Let $\mathcal{D}$ be a federation of databases, $\mathcal{P}$ be a set of (attr-safe) rules in $\mathcal{L}_Q^a$, and p be any predicate defined by $\mathcal{P}$. Let $\mathcal{P}(\mathcal{D})$ denote the output computed by $\mathcal{P}$ on input $\mathcal{D}$ and let $p^{\mathcal{P}(\mathcal{D})}$ be the relation corresponding to p in $\mathcal{P}(\mathcal{D})$. Then there exists an expression E, depending only on $\mathcal{P}$, in a-$\mathcal{SQA}$ such that $E(\mathcal{D}) = p^{\mathcal{P}(\mathcal{D})}$.*

3.5 Extended Calculus

In this section, we study a language $\mathcal{L}_C$ in the spirit of domain relational calculus that is inspired by the syntax of SchemaLog. We take a fresh look at the notion of safety in the context of $\mathcal{L}_C$ and study in depth the different levels of safety that naturally arise in the context of $\mathcal{L}_C$. We also investigate the relationship between the various safe fragments of $\mathcal{L}_C$ and the algebras of Section 3.3.

Definition 3.5.1 *A* term *of $\mathcal{L}_C$ is either a variable or a constant.* Atomic formulas *(atoms[2]) are one of the following forms:*
(i) $\langle db\rangle :: \langle rel\rangle[\, \langle attr_1\rangle \rightarrow \langle val_1\rangle, \ldots, \langle attr_n\rangle \rightarrow \langle val_n\rangle]$,
(ii) $\langle db\rangle :: \langle rel\rangle[\, \langle attr_1\rangle, \ldots, \langle attr_n\rangle]$, (iii) $\langle db\rangle :: \langle rel\rangle$, (iv) $\langle db\rangle$, where $\langle db\rangle$, $\langle rel\rangle$, $\langle attr_i\rangle$, and $\langle val_i\rangle$ are terms, or (v) an atom involving one of the built-in predicates

[2]Note that atoms in $\mathcal{L}_C$ correspond in general to molecules in $\mathcal{L}$. Also note that explicit tuple-id's are dispensed with in $\mathcal{L}_C$.

$=, <, >, \neq$. *Formulas are formed by closing atoms under the usual boolean connectives and quantifiers. Atoms of type* (i) – (iv) *are called the* **database atoms** *while those of type* (v) *are called* **built-in atoms.**

The **depth** *of a database atom in $\mathcal{L}_C$ is defined as follows. Atoms of depth 1, 2, and 3 are defined as in SchemaLog (Section 2.3.1). All other database atoms are defined to be of depth 4. Built-in atoms are of depth 0. An* **expression** *in $\mathcal{L}_C$ is of the form $\{X_1, \ldots, X_m \mid \phi(X_1, \ldots, X_m)\}$, where $X_1, \ldots, X_m$ are the distinct free variables in the $\mathcal{L}_C$ formula ϕ.*

The notions of free and bound occurrences of variables in formulas are defined as usual.

3.5.1 Domain and Safety

As $\mathcal{L}_C$ provides a formal status to database names, relation names and attribute names in the federation, our domain should include, apart from the values appearing in the federation, the names of all the databases, all the relations as well as the attribute names in them. The following definition captures this notion.

Definition 3.5.2 *Define the* **depth** *of a formula ϕ (depth(ϕ)) to be the maximum of the depth of the atoms in the formula. Let C be the set of constants appearing in ϕ. Now, the* **domain** *of ϕ denoted as $DOM(\phi)$, is defined as follows.*

If depth(ϕ) $= 0$, $DOM(\phi) = C$
If depth(ϕ) $= 1$, $DOM(\phi) = C \cup \{s \mid s$ is a database name in the federation $\}$
If depth(ϕ) $= 2$, $DOM(\phi) = C \cup \{s \mid s$ is a database name, or a relation name in the federation $\}$
If depth(ϕ) $= 3$, $DOM(\phi) = C \cup \{s \mid s$ is a database name, relation name, or an attribute name in the federation $\}$
If depth(ϕ) $= 4$, $DOM(\phi) = C \cup \{s \mid s$ is a database name, relation name, attribute name, or a value in the federation $\}$.

Safety: We would like the formulas of $\mathcal{L}_C$ that we consider, to "pay attention to the domain of the formula". Following Ullman [Ull89b], we call such domain independent formulas "safe formulas". We formally define safe formulas below. For a formula ϕ, variable X, and constant a, $\phi[a/X]$ denotes the result of replacing all free occurrences of X in ϕ by a.

Definition 3.5.3 *Let Δ denote the domain of ϕ ($DOM(\phi)$) as defined in Definition 3.5.2. A formula ϕ in $\mathcal{L}_C$ is* safe *if it satisfies the following properties.*

- *Each answer to ϕ comes from Δ.*
- *For each subformula of ϕ of the form $(\exists X)(\phi)$, $\phi[a/X]$ is false regardless of the values substituted for other free variables of ϕ, $\forall a \notin \Delta$.*
- *For each subformula of ϕ of the form $(\forall X)(\phi)$, $\phi[a/X]$ is true regardless of the values substituted for other free variables of ϕ, $\forall a \notin \Delta$.*

We call the fragment of $\mathcal{L}_C$ corresponding to the safe formulas, safe $\mathcal{L}_C$.

While the above notion of safety is theoretically sound, practical considerations, as explained below, motivate the need for taking a closer look at the notion of safety and refining it further into more specific fragments.

Alternative definitions of Domain/Safety

Consider a federation consisting of bibliographic databases all over North America. Admittedly, this federation comprises a large number of databases spread over a wide geographical landscape. Given the enormity of such a federation and the concomitant high cost of accessing information across the component databases in the federation, it is not realistic to consider expressions that refer to all the component databases, as *safe*. However, the above definition (Definition 3.5.3) of safety does not take into account such practical considerations. As an example, $\{X \mid X\}$, the expression that lists the names of all the bibliographic databases in North America is safe by the above definition!

Thus, the classical notion of safety does not suffice for the federation setting. A practically useful definition of safety should render as safe, only those expressions that restrict attention to a *specific set of databases* in the federation. Indeed, such restrictions can be extended to specific relations in known databases (and even to specific attributes in known relations). A starker and more convincing example for the need for restricting attention to subsets of the entire federation information arises in the context of the World Wide Web. When we consider the web as a federation of information sources (which we shall in Chapter 5), it is not appropriate to consider queries that refer to *all* the information sources in the web as safe. Below, we provide alternative definitions of safety that are suitable for scenarios in which the federation is huge and the cost of evaluating some classes of queries traditionally considered safe. is so high that. for practical reasons, the queries should be considered unsafe. Finally, note that if we are dealing with a federation containing a small number of component databases. when the individual access cost is not very high. then the above notion of safety is indeed appropriate.

The above observations suggest that depending on the class of applications under consideration. the applicable notion of safety should be different. This calls for a notion of varying degrees of safety - from the most liberal. in which formulas referring to all component databases in a federation should be considered safe (Definition 3.5.3), to the most restrictive in which only formulas referring to specific set of databases. relations. and even attributes should be considered safe. It follows that the different notions of safety can be obtained by modifying the notion of the domain of a formula. Indeed it is evident from Definition 3.5.3 that the safety of a formula intrinsically depends on the notion of domain. With an aim towards formulating various levels of safety. we provide alternative definitions of domain of a formula.

Definition 3.5.4 (Alternative definitions of DOM)
$DOM^0_{db}(\phi) = DOM^0_{rel}(\phi) = DOM^0_{attr}(\phi) = C$
$DOM^{i+1}_{db}(\phi) = \{s \mid \exists d \in DOM^i_{db}. s$ *is a relation name. attribute name. or value in database* $d\}$
$DOM^{i+1}_{rel}(\phi) = \{s \mid \exists d.r \in DOM^i_{rel}. s$ *is a attribute name. or value in relation* r *which is in database* $d\}$
$DOM^{i+1}_{attr}(\phi) = \{s \mid \exists d.r.a \in DOM^i_{attr}. s$ *is a value in column* a *in relation* r *which is in database* $d\}$

$$DOM_{db}(\phi) = \cup_{i \geq 0} DOM^i_{db}(\phi)$$
$$DOM_{rel}(\phi) = \cup_{i \geq 0} DOM^i_{rel}(\phi)$$
$$DOM_{attr}(\phi) = \cup_{i \geq 0} DOM^i_{attr}(\phi)$$

For each definition of domain, we have a corresponding notion of safety.

Definition 3.5.5 (Alternative definitions of safety) *A formula ϕ in $\mathcal{L}_C$ is* db-safe *(*rel-safe, attr-safe*) if it satisfies the properties of Definition 3.5.3, with Δ as $DOM_{db}(\phi)$ ($DOM_{rel}(\phi)$, $DOM_{attr}(\phi)$ respectively).*

We now state the results on the inter-relationship among the various safe fragments of $\mathcal{L}_C$. The proofs are straightforward and hence omitted.

Proposition 3.5.1 *Every db-safe $\mathcal{L}_C$ formula is a safe $\mathcal{L}_C$ formula.*

Proposition 3.5.2 *Every rel-safe $\mathcal{L}_C$ formula is a db-safe $\mathcal{L}_C$ formula.*

Proposition 3.5.3 *Every attr-safe $\mathcal{L}_C$ formula is a rel-safe $\mathcal{L}_C$ formula.*

Our next result demonstrates that the expressive power of $\mathcal{L}_C$ is no less than that of $\mathcal{SQA}$.

Lemma 3.5.1 *Every expression of $\mathcal{SQA}$ is expressible in safe $\mathcal{L}_C$.*

PROOF. The proof is an induction on the number of operators in the $\mathcal{SQA}$ expression, say E. It is very similar to the proof of expressibility of classical algebra expressions in safe DRC ([Ull89b]). The only difference is that we have one new base case (δ), and three new induction cases (ρ, α, γ) to be considered.

Base Case: $E = \delta()$: The safe $\mathcal{L}_C$ formula corresponding to this expression is $\{X \mid X\}$. The safety of this formula follows from the definition.

Induction:

Case 1. $\rho(E_1)$: Let E_1 be equivalent to the safe query $\{D \mid \phi(D)\}$. Then E is equivalent to the safe query $\{D, R \mid \phi(D) \wedge D :: R\}$.

Case 2. $\alpha(E_1)$: Let the safe query corresponding to E_1 be $\{D, R \mid \psi(D, R)\}$. E is then equivalent to the safe query $\{D, R, A \mid \psi(D, R) \wedge D :: R[A]\}$.

Case 3. $\gamma_{\langle \rightarrow ,a_2 \rightarrow ,\ldots, \rightarrow v_n \rangle}(E_1)$: Let E_1 be equivalent to the safe $\mathcal{L}_C$ query $\{D, R \mid \chi(D, R)\}$. Then E is equivalent to the safe query,
$\{D, R, A_1, V_1, A_2, V_2, \ldots, A_n, V_n \mid \chi(D, R) \wedge D :: R[A_1 \rightarrow V_1, A_2 \rightarrow V_2, \ldots, A_n \rightarrow V_n] \wedge A_2 = a_2 \wedge V_n = v_n\}$.

The safety of the equivalent $\mathcal{L}_C$ queries is straightforward. □

We now turn our attention to the expressibility of the other fragments of the calculus by the corresponding fragments of the algebra.

Lemma 3.5.2 *Every d-$\mathcal{SQA}$ expression can be expressed by a db-safe $\mathcal{L}_C$ formula.*

PROOF. Recall that *d-$\mathcal{SQA}$* is obtained from $\mathcal{SQA}$ by replacing the $\delta()$ operation with the $\hat{\delta}(s)$ operation. Thus we can obtain our proof from the proof of Lemma 3.5.1, by showing how $\hat{\delta}(s)$ can be expressed using a db-safe $\mathcal{L}_C$ formula. We present the modifications necessary to the above proof.

The new base case for the proof is the scenario in which there are zero operators in the *d-$\mathcal{SQA}$* expression E. Then E is a set consisting of constants $\{c_1, \ldots, c_n\}$. The equivalent db-safe $\mathcal{L}_C$ formula for E is $\{X \mid X = c_1 \vee \ldots \vee X = c_n\}$.

A new inductive case arises for the *d-$\mathcal{SQA}$* operator $E = \hat{\delta}(E_1)$: Let E_1 be equivalent to the db-safe $\mathcal{L}_C$ query $\{D \mid \phi(D)\}$. Then E is the db-safe $\mathcal{L}_C$ query $\{D \mid \phi(D) \wedge D\}$.

The rest of the inductive cases are identical to those in the proof of Lemma 3.5.1. A crucial observation that can be easily verified is that the equivalent safe $\mathcal{L}_C$ queries for these cases are also db-safe. □

Expressibility results similar to the ones above, extend to the other fragments of the algebra and calculus languages.

Lemma 3.5.3 *Every r-$\mathcal{SQA}$ expression can be expressed by a rel-safe $\mathcal{L}_C$ formula.*

Lemma 3.5.4 *Every a-$\mathcal{SQA}$ expression can be expressed by a attr-safe $\mathcal{L}_C$ formula.*

The proofs of Lemmas 3.5.3 and 3.5.4 are similar to the proof for the Lemma 3.5.2 extended to the *rel* and *attr* fragments respectively.

We now turn our attention to the relative expressive power of the $\mathcal{L}_C$ and $\mathcal{L}_Q$ languages. Our first result in this direction states that the expressive power of safe $\mathcal{L}_Q$ language is no less than that of the safe $\mathcal{L}_C$ fragment.

Lemma 3.5.5 *Every safe $\mathcal{L}_C$ query can be expressed in safe $\mathcal{L}_Q$.*

PROOF. This proof works along the lines of the proof of expressibility of safe DRC queries in safe, non-recursive Datalog [Ull89b].

It can be shown that for every safe $\mathcal{L}_C$ query $\{\bar{X} \mid \phi(\bar{X})\}$, there is an equivalent (safe) $\mathcal{L}_C$ query $\{\bar{X} \mid \psi(\bar{X})\}$, where the formula ψ satisfies the following conditions.

- ψ does not contain any use of $\forall$.

- If $F_1 \vee F_2$ is a subformula in ψ, F_1 and F_2 have the same set of free variables.
- If $F_1 \wedge \cdots \wedge F_n$ is a maximal conjunct in ψ, then all free variables in F_i are *limited* by *(a)* appearing free in F_j ($j = i$ possibly) where F_j is not a built-in atom and is not a negated formula, or *(b)* being equated to a constant or a limited variable (perhaps through a chain of equalities).
- Whenever ψ has a subformula $\neg\varphi$, $\neg\varphi$ is part of a subformula of the form $\varphi_1 \wedge \cdots \wedge \varphi_k \wedge \neg\varphi \wedge \varphi_{k+1} \wedge \cdots \wedge \varphi_m$, where at least one of the φ_i's is not negated.

Indeed, ϕ can be translated to ψ algorithmically, as discussed in [Ull89b]. Let F be any safe $\mathcal{L}_C$ formula. By the above, we may assume without loss of generality that F satisfies the above conditions.

Let G be a maximal conjunct of subformulas of F. Let $X_1, \ldots, X_n$ be the free variables in G. We prove that for every subformula G, there is a $\mathcal{L}_Q$ program that defines a relation for some programming predicate $p_G(X_1, \ldots, X_n)$, such that $p_G(a_1, \ldots, a_n)$ is true *iff* $G[a_1/X_1, \ldots, a_n/X_n]$ is true. Here $G[a_1/X_1, \ldots, a_n/X_n]$ denotes the ground formula obtained by substituting a_i for X_i in G.

Let $G \equiv G_1 \wedge \cdots \wedge G_k$. The base case is when $k = 1$ and G_1 is one of the $\mathcal{L}_C$ atoms. We define a predicate p_G for G by $p_G(X_1, \ldots, X_m) \longleftarrow G_1 \wedge \cdots \wedge G_k$, where $X_1, \ldots, X_m$ are the free variables in G. From the definition of safe $\mathcal{L}_C$ formulas, it follows that X_i's are limited. This is thus a safe rule in $\mathcal{L}_Q$.

Induction: We need to consider three cases - $\exists$, $\vee$, and $\wedge$. G does not contain $\forall$, and $\neg$ can only appear within conjunctions.

($\exists$) Let $G = (\exists X_i)H$, where $X_1, \ldots, X_k$ are the free variables in the atom H. The predicate corresponding to p_G can be defined as

$p_G(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_k) \longleftarrow p_H(X_1, \ldots, X_k)$.

($\vee$) Let $G = H \vee I$. By the definition of safety, free variables of H and I must be the same. The proof of this claim would be based on the argument that if I has some free variable that does not appear in H, whenever H is true, I need not be true, and hence this free variable can take on any value (in particular, one that does not belong to DOM). Let the free variables in H (and I) be $X_1, \ldots, X_k$. The following two rules can be used to express G.

$p_G(X_1, \ldots, X_k) \longleftarrow p_H(X_1, \ldots, X_k)$

$p_G(X_1, \ldots, X_k) \longleftarrow p_I(X_1, \ldots, X_k)$

($\wedge$) Let $G = G_1 \wedge \cdots \wedge G_n$. The rule for G can be expressed as: $p_G(X_1, \ldots, X_k) \longleftarrow S_1 \wedge \cdots \wedge S_n$, where S_i is the subgoal corresponding to G_i (obtained inductively) and $X_1, \ldots, X_k$ are the free variables appearing among the G_i's. □

Similar result extends to the other fragments of the respective languages as well, and below we present these results.

Lemma 3.5.6 *Every db-safe $\mathcal{L}_C$ query can be expressed using a $\mathcal{L}_Q^d$ program.*

PROOF. The proof of this lemma differs little from that of Lemma 3.5.5; in particular, the only modification required is to prove that the free variables in a maximal conjunct of subformulas of any db-safe $\mathcal{L}_C$ formula, are db-limited. This result follows from the definition of db-safe $\mathcal{L}_C$ formulas. The rest of the proof can be obtained along the lines of the proof of Lemma 3.5.5. □

The proofs of the following two lemmas are similar to that of Lemma 3.5.6, extended to the rel and attr fragments respectively.

Lemma 3.5.7 *Every rel-safe $\mathcal{L}_C$ query can be expressed using a $\mathcal{L}_Q^r$ program.*

Lemma 3.5.8 *Every attr-safe $\mathcal{L}_C$ query can be expressed using a $\mathcal{L}_Q^a$ program.*

We are now ready to present our final result of this section that establishes the equivalence of the various languages introduced so far.

Theorem 3.5.1 *The set of queries expressible in each of the following sets of languages is identical.*

- $\mathcal{SQA}$, safe $\mathcal{L}_C$, and safe $\mathcal{L}_Q$
- *d-*$\mathcal{SQA}$, db-safe $\mathcal{L}_C$, and $\mathcal{L}_Q^d$
- *r-*$\mathcal{SQA}$, rel-safe $\mathcal{L}_C$, and $\mathcal{L}_Q^r$
- *a-*$\mathcal{SQA}$, attr-safe $\mathcal{L}_C$, and $\mathcal{L}_Q^a$

PROOF. Follows from Lemmas 3.4.1 - 3.5.8. □

While the proof of equivalent expressive power of these languages, presented above is roundabout, in Appendix B, we present some direct results on the relative expressive powers of the various pairs of languages.

3.6 Restructuring Algebra

In this section, we introduce the type (3) operators discussed in the beginning of this chapter. While the operators of Section 3.3 are capable of data and schema querying, the operators we introduce below query as well as *restructure* data and meta-data in a federation. Thus, these operators are useful for realizing the restructuring functionality of SchemaLog. We will denote the algebra consisting of the restructuring operators defined in this section as $\mathcal{SRA}$. By $\mathcal{SA}$, we denote the algebra consisting of both the $\mathcal{SQA}$ and $\mathcal{SRA}$ operators. We now present the $\mathcal{SRA}$ operators.

Definition 3.6.1 (Creating databases) *The first operator $\boldsymbol{\vartheta}$ takes a programming relation of arity k as input and a column number $i \leq k$ as a parameter, and creates databases with names corresponding to the entries appearing in column i of the input relation. More formally, $\boldsymbol{\vartheta}_i(s)$, for a programming relation s and a column number i creates a database named d for each $d \in \pi_i(s)$, if such a database does not already exist.*

For example, $\boldsymbol{\vartheta}_1(\{(univA), (univB), (univC)\}$ creates the databases $univA, univB, univC$ (which do not contain any other information).

Definition 3.6.2 (Creating relations in databases) *The second operator $\boldsymbol{\kappa}$ takes a programming relation of arity k as input and two column numbers $i, j \leq k$ as parameters. It creates databases with names corresponding to the entries appearing in column i (of the input relation), and containing relations whose names are obtained by interpreting the entries appearing in column j. More formally, $\boldsymbol{\kappa}_{i,j}(s)$, for a programming relation s and column numbers i, j creates a database named d with relations having names $r_1, \ldots, r_n$ exactly when $\sigma_{\$1='d'}(\pi_{i,j}(s)) = \{(d, r_1), \ldots, (d, r_n)\}$, whenever such a database does not already exist. Note that $\sigma_{\$1='d'}$ denotes classical selection.*

For example, $\boldsymbol{\kappa}_1(\{(univA, payInfo), (univC, cs), (univC, ece)\})$ creates the databases $univA$ and $univC$, the former containing a relation $payInfo$, and the latter containing the relations cs and ece. (The scheme of the relations is not yet defined.)

Definition 3.6.3 (Creating relations with schemas) *The next operator ς takes a programming relation of arity k as input and three column numbers $i, j, l \leq k$ as parameters. It creates databases with names corresponding to the entries in column i,* *having relations with names corresponding to entries in column j and whose schemas are determined by interpreting the entries appearing column l as the attributes associated with the relation names in column j. More formally, for a programming relation s and column numbers i, j, l, $\varsigma_{i,j,l}(s)$ creates a database d containing a relation r with attributes $a_1, \ldots, a_n$ exactly when $\sigma_{\$1='d'}(\pi_{i,j,l}(s)) = \{(d, r, a_1), \ldots, (d, r, a_n)\}$, whenever such a database does not already exist.*

For example, let $s = \{(univC, cs, category), (univC, ece, category)\}$. Then, $\varsigma_{1,2,3}(s)$ will create a database $univC$ containing two relations cs and ece both with the schema $\{catergory\}$ and with no data.

Definition 3.6.4 (Creating and populating relations with schemas) *The last operator ϱ takes a programming relation of arity k as input, four column numbers i, j, k, l and another list of column numbers $g_1, \ldots, g_m$ as parameters and returns as output several databases structured according to the interpretation of column i entries as database names, column j entries as relation names, column k entries as attribute names, and column l entries as values. Facts so generated form pieces of larger tuples. The grouping for forming output tuples is determined based on equality on the columns $g_1, \ldots, g_m$. More formally, $\varrho_{i,j,k,l:g_1,\ldots,g_m}(s)$ creates a database d containing relation r with attributes $a_1, \ldots, a_n$ exactly when $\sigma_{\$1='d'}(\pi_{i,j,k}(s)) = \{(d, r, a_1), \ldots, (d, r, a_n)\}$, whenever such a database does not already exist. Furthermore, it populates the relation r of database d with a tuple t such that $t[a_1, \ldots, a_n] = \langle v_1, \ldots, v_n \rangle$ exactly when $\exists t_1, \ldots, t_n \in s$ such that $\pi_{i,j,k,l}(\{t_1, \ldots, t_n\}) = \{(d, r, a_1, v_1), \ldots, (d, r, a_n, v_n)\}$, and finally, $t_1[g_1, \ldots, g_m] = \cdots = t_n[g_1, \ldots, g_m]$. When the relation r already exists, the above mentioned tuples are appended to this relation.*

For example, let r be the relation:

$\{(univC, cs, avg, prof, 65K),$
$(univC, cs, avg, Aprof, 40K),$
$(univC, cs, cat, prof, prof),$
$(univC, cs, cat, Aprof, Aprof),$
$(univC, ece, avg, sec, 30K),$
$(univC, ece, avg, prof, 70K),$
$(univC, ece, cat, prof, prof),$
$(univC, ece, cat, sec, sec)\}$

Then, $\varrho_{1,2,3,5:4}(r)$ will create the database shown in Figure 6.

univ-C

cs

cat	avg
Prof	65,000
AProf	40,000
.	.

ece

cat	avg
Sec	30,000
Prof	70,000
.	.

Figure 6: Example of Restructuring

We remark that our restructuring operations also have limited update capabilities in the sense that whenever the relation corresponding to the restructured form already exists, newly generated data is appended to such a relation. E.g., suppose that the relation *payInfo* shown in Figure 6 already exists (say, because of an invocation of operation ϱ).
Let $s' = \{(univC, cs, avg, sec, 35K), (univC, cs, cat, sec, sec)\}$. Then $\varrho_{1,2,3,5;4}(s')$ will have the effect of *appending* the tuple $(sec, 35K)$ to the *existing* cs relation of database $univC$.

We now present an example algebraic program, that illustrates the expressive power of our (querying and restructuring) algebra, $\mathcal{SA}$.

Example 3.6.1 *Suppose it is required to restructure the information in the* univC *database of our running example such that it conforms to the schema of the* univB *database. The intended effect is that the there would be a single relation (called, say* payInfo*) whose attributes are* category *and the names of the departments in* univC*. The department columns contain the* avgSal *information. The following algebraic expression accomplishes this restructuring in a straightforward manner.*

$$\varrho_{1,5,2,4;3}((\pi_{1,3,4,4}(\gamma_{\langle cat\rightarrow\ ,avg\rightarrow\ \rangle}(\rho(\{(univC)\})))$$
$$\cup\pi_{1,2,4,6}(\gamma_{\langle cat\rightarrow\ ,avg\rightarrow\ \rangle}(\rho(\{(univC)\})))) \times \{(payInfo)\})$$

For the sake of clarity, we explain the details of computation of the above algebraic expression in the form of a sequence of simple steps showing intermediate results. For this purpose, we use temporary (programming) relations to store intermediate results.

1. $r_1 := \boldsymbol{\rho}(\{(univC)\})$
2. $r_2 := \gamma_{\langle cat\longrightarrow\ ,avg\longrightarrow\ \rangle}(r_1)$
3. $r_3 := \pi_{1,2,4,6}(r_2)$
4. $r_4 := \pi_{1,3,4,4}(r_2)$
5. $r_5 := r_3 \cup r_4$
6. $r_6 := r_5 \times \{(payInfo)\}$
7. $r_7 := \varrho_{1,5,2,4:3}(r_6)$

The steps are self-explanatory.

3.7 Discussion and Conclusion

As suggested by Example 3.6.1 of the preceding section. $\mathcal{SA}$ can be regarded as the algebra that lies at the heart of an implementation of SchemaLog. In [ALNI96], we study this issue in detail and propose an implementation architecture that is based on compiling SchemaLog constructs into $\mathcal{SA}$. We also address the challenging issues unique to the SchemaLog implementation and propose three alternative storage structures for dealing with them. Following up this work, [Ala97] proposes algorithms for top-down implementation of SchemaLog, including alternative strategies for the implementation of the algebraic operators. This work also evaluates the effectiveness of the alternate strategies with a series of experiments on real-life databases running on MS Access. These works conclude that from practical considerations. a viable approach for implementing SchemaLog programs seems to be to use conventional storage for existing database relations and a storage strategy known as reduced storage (based on the first-order reduction result of SchemaLog) [ALNI96] for derived database relations. We refer to [ALNI96. Ala97] for the details.

An important question that arises in the context of such an implementation is the *sufficiency* of $\mathcal{SA}$ to express every program in the full-fledged SchemaLog language. In particular. the full SchemaLog language has function symbols which are used in an essential way for performing advanced database programming (as illustrated in Section 2.6.1). Note that $\mathcal{SA}$ captures the functionality of function symbols by means of the grouping parameters set $\{g_1, \ldots, g_n\}$ of the ϱ operator: the 'assembling' of the output tuples is determined based on the equality on these columns. In particular.

there is no order associated with the parameters $\{g_1, \ldots, g_n\}$. However, note that function symbols in SchemaLog have an implicit ordering associated with them.

We remark that the lack of such an ordering on the grouping attributes exhibited by the algebra, can be imposed on SchemaLog by introducing equality axioms that capture this notion. Conversely, $\mathcal{SA}$ can be developed along the SchemaLog direction by introducing function symbols (or a means by which they can be simulated) in the algebra. However, we do not pursue these issues further in this thesis. As a final remark, as illustrated by the various results in this chapter, $\mathcal{SA}$ can be seen to be sufficiently expressive to cater to most of the commonly occuring SchemaLog programs.

Chapter 4

SchemaSQL

Since its inception in IBM's System R database management system, SQL ([AC75, Cha76]) has been the reigning relational database query language supported by most database management systems. It is the lingua franca of the database community, especially for the practitioners in the industry. While SchemaLog is a powerful language with formal theoretical basis, it is difficult for a typical database practitioner well-oiled in SQL, to adapt to its syntax, and start developing applications in SchemaLog. On the other hand, it will be easier for the same user to adapt to a language having a syntax closer to that of SQL. Motivated by this practical need, we develop an SQL-like language that derives its inspirations from SchemaLog. In particular, in the spirit of SchemaLog, our language (a) has a expressive power that is independent of the federation/database schema, (b) is easy to use, yet sufficiently expressive, (c) is capable of performing data and schema querying as well as restructuring, and (d) has the ability to perform complex of aggregation beyond the realm of what is possible in standard SQL. Besides, in view of the importance and popularity of SQL in the database world, our language provides full data manipulation and view definition capabilities of SQL and is *downward compatible* with SQL, in the sense that it is compatible with SQL syntax and semantics. We call this dialect of SQL, *SchemaSQL* .

SchemaSQL is not a language obtained by extending SQL with ad-hoc constructs capable of mimic-ing the features of SchemaLog. On the contrary, we have paid careful attention to the study of the formal semantics of standard SQL and on how the semantics can be extended to *SchemaSQL* in a natural and intuitively pleasing manner. The end result is a language that is easy for the SQL user to adapt. Also, as a result of the principled extension, *SchemaSQL* queries can be processed by reducing them to a series of SQL queries, thus enabling the possibility of realizing an implementation of *SchemaSQL* on top of existing DBMS infrastructures. This chapter discusses all these issues in depth, and is organized as follows.

4.1 Overview of the Chapter

(1) We review the syntax and semantics of SQL, and develop *SchemaSQL* as a *principled extension* of SQL (Sections 4.2, 4.3.2). As a result, for a SQL user, adapting to *SchemaSQL* is relatively easy. (2) We illustrate via examples the following powerful features of *SchemaSQL* : (i) *uniform* manipulation of data and meta-data; (ii) creating *restructured views* and the ability to *dynamically create output schemas*; (iii) the ability to express *sophisticated aggregate computations* far beyond those expressible in conventional languages like SQL (Sections 4.3.3, 4.4.2). (3) We propose an *implementation architecture* for *SchemaSQL* that is designed to build on *existing* RDBMS technology, and requires *minimal additions* to it, while greatly enhancing its power (Section 4.5). We provide an implementation algorithm for *SchemaSQL*, and establish its correctness. We also discuss novel query optimization issues that arise in the context of this implementation. (4) Finally, we propose an extension to *SchemaSQL* for systematically resolving the *semantic heterogeneity* problem arising in a MDBS environment (Section 4.6).

4.2 Syntax

Our goal is to develop *SchemaSQL* as a principled extension of SQL. To this end, we briefly analyze the syntax of SQL, and then develop the syntax of *SchemaSQL* as a natural extension. Our discussion below is itself a novel way of viewing the syntax and semantics of SQL, which, in our opinion, helps a better understanding of SQL subtleties.

In an SQL query, the (tuple) variables are declared in the `from` clause. A variable declaration has the form `<range> <var>`. For example, in the query in Figure 7(a), the expression `emp T` declares `T` as a variable that ranges over the (tuples of the) relation `emp` (in the usual SQL jargon, these variables are called *aliases*.) The `select` and `where` clauses refer to (the extension of) attributes, where an attribute is denoted as **<var>.<attName>**, **var** being a (tuple) variable declared in the **from** clause, and **attName** being the name of an attribute of the relation (extension) over which **var** ranges.

```
select T.name         select emp.name        select name
from   emp T          from   emp             from   emp
where  T.dept =       where  emp.dept =      where  dept =
         "Marketing"           "Marketing"            "Marketing"
      (a)                   (b)                     (c)
```

Figure 7: Syntax of Simple SQL Queries

When no ambiguity arises, SQL permits certain abbreviations. Queries of Figure 7(b,c) are equivalent to the first one, and are the most common ways such queries are written in practice. Note that in Figures 7(b) and 7(c), **emp** acts essentially as a tuple variable.

The *SchemaSQL* syntax extends that of SQL in several directions.

1. The federation consists of databases, with each database containing relations. The syntax allows to distinguish between (the components of) different databases.

2. To permit meta-data queries and restructuring views, *SchemaSQL* permits the declaration of other types of variables in addition to the (tuple) variables permitted in SQL.
3. Aggregate operations are generalized in *SchemaSQL* to make horizontal and block aggregations possible. in addition to the usual vertical aggregation in SQL.

In this section we will concentrate on the first two aspects. Restructuring views and aggregation are discussed in Section 4.4.

Variable Declarations in *SchemaSQL*

SchemaSQL permits the declaration of variables that can range over any of the following five sets: (i) names of databases in a federation; (ii) names of the relations in a database; (iii) names of the attributes in the scheme of a relation; (iv) tuples in a given relation in a database; and (v) values appearing in a column corresponding to a given attribute in a relation. Variable declarations follow the same syntax as **<range> <var>** in SQL, where **var** is any identifier. However, there are two major differences. (1) The only kind of range permitted in SQL is a set of tuples in some relation in the database, whereas in *SchemaSQL* any of the five kinds of ranges above can be used to declare variables. (2) More importantly, the range specification in SQL is made using a constant, i.e. an identifier referring to a specific relation in a database. By contrast, the diversity of ranges possible in *SchemaSQL* permits range specifications to be *nested*, in the sense that it is possible to say, e.g., that `X` is a variable ranging over the relation names in a database `D`, and that `T` is a tuple in the relation denoted by `X`. These ideas are made precise in the following definition.

Definition 4.2.1 (Range Specifications) *The concepts of* range specifications, constant, *and* variable *identifiers are simultaneously defined by mutual recursion as follows:*

1. Range specifications *are one of the following five types of expressions, where* `db`, `rel`, `attr` *are any constant or variable identifiers (defined in 2 below).*

   (a) The expression `->` *denotes a range corresponding to the set of database names in the federation.*

   (b) The expression `db->` *denotes the set of relation names in the database* `db`.

   (c) The expression `db::rel->` *denotes the set of names of attributes in the scheme of the relation* `rel` *in the database* `db`[1].

   (d) `db::rel` *denotes the set of tuples in the relation* `rel` *in the database* `db`.

   (e) `db::rel.attr` *denotes the set of values appearing in the column named* `attr` *in the relation* `rel` *in the database* `db`.

2. A variable declaration is of the form `<range> <var>` *where* `<range>` *is one of the range specifications above and* `<var>` *is an identifier*[2]*.. An identifier* `<var>` *is said to be a* variable *if it is declared as a variable by an expression of the form* `<range> <var>` *in the* `from` *clause. Variables declared over the ranges (a) to (e) are called* db-name, rel-name, attr-name, tuple, *and* domain variables, *respectively. Any identifier not so declared is a* constant.

As an illustration of the idea of nesting variable declarations, consider the clause `from db1-> X, db1::X T`. This declares `X` as variable ranging over the set of relation names in the database `db1` and `T` as a variable ranging over the tuples in each relation `X` in the database `db1`.

[1]The intuition for the notation is that we can regard the attributes of a relation as written to the right of the relation name itself!

[2]Abbreviations similar in spirit to those allowed for SQL are also allowed in *SchemaSQL* .

We will use the University federation of Chapter 1, expanded with yet another database component univ-D (shown in Figure 8) as the running example for this chapter.

`univ-A`

salInfo

category	dept	salFloor
Prof	CS	65.000
Assoc Prof	CS	50,000
Prof	Math	60,000
Assoc Prof	Math	55,000

`univ-B`

salInfo

category	CS	Math
Prof	55,000	65,000
Assoc Prof	50,000	55,000

`univ-D`

salInfo

dept	Prof	Assoc Prof
CS	75,000	60.000
Math	60,000	45.000

`univ-C`

CS

category	salFloor
Prof	60.000
Assoc Prof	55,000

Math

category	salFloor
Prof	70,000
Assoc Prof	60,000

Figure 8: Representing Similar Information Using Different Schemas in Multiple Databases univ-A, univ-B, univ-C, and univ-D

Example 4.2.1 List the departments in `univ-A` that pay a higher salary floor to their technicians compared with the same department in `univ-B`.

```
select A.dept
from   univ-A::salInfo A, univ-B::salInfo B,
       univ-B::salInfo-> AttB
where       AttB        <> "category"   and
            A.dept      =  AttB         and
(Q1)        A.category  =  "technician" and
            B.category  =  "technician" and
            A.salFloor  >  B.AttB
```

Explanation: Variables `A` and `B` are (SQL-like) tuple variables ranging over the relations `univ-A::salInfo` and univ-B::salInfo, respectively. The variable AttB is declared as an attribute name of the relation univ-B::salInfo. It is intended to be a dept_i attribute (hence the condition AttB <> "category" in the where clause). The rest of the query is self-explanatory. ■

Example 4.2.2 List the departments in univ-C that pay a higher salary floor to their technicians compared with the same department in univ-D.

```
        select RelC
        from   univ-C-> RelC, univ-C::RelC C,
               univ-D::salInfo D
(Q2)    where  RelC        = D.dept       and
               C.category = "technician" and
               C.salFloor > D.technician
```

Explanation: The variable RelC is declared as a relation name in the database univ-C. Note that in this database there is one relation per department, and the relation name coincides with department name. Variable C is then declared as a tuple variable on this (variable) relation RelC. The variable D is an (SQL-like) tuple variable ranging over the relation univ-D::salInfo. Note that in univ-D::salInfo categories are represented by attribute names, whose domains consist of the salary floors of the corresponding category. Hence, D.technician is the salary floor for category technician (for the tuple represented by tuple variable D). ■

4.3 Semantics I: Fixed Output Schema

We first quickly review the semantics of SQL and express it in a manner that makes it possible to realize the semantics of *SchemaSQL* by a simple extension.

4.3.1 SQL Semantics Reviewed

A query in SQL assumes a *fixed* scheme for the underlying database, and maps each database to a relation over a fixed scheme, called the *output* scheme associated with the query. Let $\mathcal{D}$ be the set of all database instances over a fixed scheme. Let a query Q be of the form

```
select    attrList, aggList
from      fromList
where     whereConditions
group by  groupbyList
having    haveConditions
```

Let $\mathcal{R}$ be the set of all relations over the output scheme of the query Q. The query Q induces a function

$$Q : \mathcal{D} \rightarrow \mathcal{R}$$

from databases to relations over a fixed scheme. defined as follows. Let $D \in \mathcal{D}$ be an input database. and $\mathcal{T}_D$ the set of all tuples appearing in any relation in D. Let τ be the set of tuple variables occurring in Q. We define an *instantiation* as a function $\imath : \tau \rightarrow \mathcal{T}_D$ which instantiates each tuple variable in Q to some tuple over its appropriate range. The conditions whereConditions in the where clause induce a boolean function. denoted $sat(\imath, Q)$, on the set of all instantiations, reflecting whether the conditions are satisfied by an instantiation. This is defined in the obvious manner. Let $\mathcal{I}_Q = \{\imath \mid \imath \text{ is an instantiation for which } sat(\imath, Q) = true\}$ denote the set of instantiations satisfying the conditions in the where clause. The query assembles each satisfying instantiation into a tuple for the answer relation, as follows. Let $\mathcal{T}_{\texttt{attrList}}$ denote the set of all tuples over the scheme attrList such that each value in each tuple appears in the database D. Then the tuple assembly function is a function $tuple_Q : \mathcal{I}_Q \rightarrow \mathcal{T}_{\texttt{attrList}}$ defined as follows.

$$tuple_Q(\imath) = \bigotimes_{\text{"}t.A\text{"} \in \texttt{attrList}} \imath(t)[A]$$

Here. the predicate "$t.A$" $\in$ attrList indicates the condition that the attribute denotation $t.A$ literally[3] appears in the list of attributes attrList in the select statement. The symbol $\otimes$ denotes concatenation. and $\imath(t)[A]$ denotes the restriction of the tuple $\imath(t)$ to the attribute A. For an instantiation $\imath$, $tuple_Q(\imath)$ produces a tuple over the attributes attrList listed in the select statement. Suppose Q is a regular query. *i.e.* a query without aggregation. In this case, the aggList is empty and the having and group by clauses are absent, and the result of the query is captured by the function

$$Q(D) = \{tuple_Q(\imath) \mid \imath \in \mathcal{I}_Q\}$$

[3]Modulo the abbreviations permitted in SQL. as explained in the beginning of Section 4.2.

To account for aggregation, we need the following extension. We define a relation $\sim$ on the instantiations.

Definition 4.3.1 *For* $\imath, \jmath \in \mathcal{I}_Q$, $\imath \sim \jmath$ *iff* $\forall$"$t.A$" $\in$ groupbyList, $\imath(t)[A] = \jmath(t)[A]$. *It is straightforward to see that* $\sim$ *is an equivalence relation on* $\mathcal{I}_Q$. *This definition essentially says that two instantiations (satisfying the conditions in the* where *clause) are* $\sim$*-equivalent provided they agree on all attributes appearing in the* group by *clause.*

Let $\mathcal{T}_{\texttt{aggList}}$ denote the set of tuples over the scheme aggList. We define a function $aggregate_Q : \mathcal{I}_Q \rightarrow \mathcal{T}_{\texttt{aggList}}$ as follows.

$$aggregate_Q(\imath) = \bigotimes_{\text{“}agg_B(t.B)\text{”} \in aggList} agg_B([\jmath(t)[B] \mid \jmath \in \mathcal{I}_Q, \jmath \sim \imath])$$

For a given instantiation $\imath$, $aggregate_Q$ considers all instantiations equivalent to $\imath$, and, for each aggregate operation, say agg_B, indicated on the attribute $t.B$ in the aggList, it performs the operation agg_B on the *multiset* of values associated with this attribute by any instantiation equivalent to $\imath$. We use $[\ldots]$ instead of $\{\ldots\}$ to denote multisets.

Now, we are ready to describe the tuple assembly associated with aggregate queries. Let Q be a query involving aggregation. We define a function $aggtuple_Q : \mathcal{I} \rightarrow \mathcal{T}_{\texttt{attrList}} \times \mathcal{T}_{\texttt{aggList}}$ as follows.

$$aggtuple_Q(\imath) = tuple_Q(\imath) \bigotimes aggregate_Q(\imath)$$

Finally, the result of an aggregate query is captured by the function

$Q(D) = \{aggtuple_Q(\imath) \mid \imath \in \mathcal{I}_Q$ & the tuple $aggtuple_Q(\imath)$ satisfies the conditions haveConditions in the having clause$\}$.

Notice that in addition to assembling tuples in accordance with the function $aggtuple_Q$, a filter is applied to check whether the conditions in the having clause are satisfied, as such conditions may involve aggregate values.

4.3.2 Semantics of *SchemaSQL* Queries

The semantics of *SchemaSQL* is obtained as a natural extension of that of SQL. A *SchemaSQL* query Q is of the form:

```
select  itemList, aggList
from    fromList
```

```
where    whereConditions
group by groupbyList
having   haveConditions
```

where itemList is a list of db-name, rel-name, attr-name, and domain variables; aggList is a list of expressions of the form agg(X) where agg is an aggregate function, and X is a variable declared in the from clause; fromList is a list of variable declarations; groupbyList is a list of variables; and the conditions in the where and having clauses are analogous to SQL[4]. The main difference with an SQL query is the availability of additional variable types (in addition to the usual SQL tuple variables).

Let $\mathcal{D}$ be the set of all federation database instances. Let $\mathcal{R}$ be the set of all relations over the output scheme of the query Q. The query Q induces a function

$$Q : \mathcal{D} \rightarrow \mathcal{R}$$

from federations to relations, defined as follows. Let $D \in \mathcal{D}$ be an input federation, and $\mathcal{O}_D$ the set of all items (database names, relation names, attribute names, tuples, and values) appearing in D. Let V be the set of variables occurring in Q. We define an *instantiation* as a function $\imath : V \rightarrow \mathcal{O}_D$ which instantiates each variable in Q to some item over its appropriate range. *Throughout this chapter, we assume that any instantiation $\imath$ is extended in such a way that for a literal constant c, $\imath(c) = c$.* In defining the semantics of *SchemaSQL* queries, we will find the following definitions useful. Identifiers in typewrite font (*e.g.* db) can be constants or variables.

Definition 4.3.2 Admissibility
An instantiation $\imath$ is an admissible *provided, it satisfies the following conditions.*

- *whenever* db-> R *is a declaration in the* from *clause,* $\imath$(R) *is the name of a relation in the database* $\imath$(db).

- *whenever* db::rel-> A *is a declaration in the* from *clause,* $\imath$(A) *is an attribute name in the relation* $\imath$(rel) *in the database* $\imath$(db).

- *whenever* db::rel T *is a declaration in the* from *clause,* $\imath$(T) *is a tuple in the relation* $\imath$(rel) *in the database* $\imath$(db).

[4]Abbreviations similar in spirit to those allowed for SQL (see Section 4.2) are also allowed in *SchemaSQL* .

- *whenever* db::rel.attr V *is a declaration in the* from *clause,* $\imath$(V) *is a value that appears in the column* $\imath$(attr) *of the relation* $\imath$(rel) *in the database* $\imath$(db).

Definition 4.3.3 Validity
Let $sat(\imath, Q)$ *be a boolean function on the set of all instantiations, induced by the conditions in the* where *clause. An instantiation* $\imath$ *is* valid *provided* (a) $sat(\imath, Q)$ *is true, and* (b) *whenever* T.attr V *is a variable declaration in the* from *clause,* $\imath(\mathrm{V}) = \imath(\mathrm{T})[\imath(\mathtt{attr})]$.

Definition 4.3.2 precisely captures the notion of an appropriate range for a variable. The conditions in the where clause as well as additional conditions induced by the presence of certain patterns involving implicit or explicit tuple variables is captured in Definition 4.3.3. We now define.

$\mathcal{I}_Q = \{\imath \mid \imath$ is a admissible and valid instantiation$\}$

The query assembles each satisfying instantiation into a tuple for the answer relation, as follows. Let $\mathcal{T}_{\mathtt{itemList}}$ denote the set of all tuples over the scheme itemList such that each value in each tuple appears in the federation D. Then the tuple assembly function is a function $tuple_Q : \mathcal{I}_Q \rightarrow \mathcal{T}_{\mathtt{itemList}}$ defined as follows.

$$tuple_Q(\imath) = \bigotimes_{s \in \mathtt{itemList}} \imath(s)$$

where s is a db-name, rel-name, attr-name, or domain variable. The symbol $\otimes$ denotes concatenation. For an instantiation $\imath$, $tuple_Q(\imath)$ produces a tuple over the list of objects itemList listed in the select statement. Suppose Q is a query without aggregation. In this case the result of the query is captured by the function

$$Q(D) = [tuple_Q(\imath) \mid \imath \in \mathcal{I}_Q]$$

Similar to SQL, *SchemaSQL* 's semantics is based on multisets. Multisets are distinguished from sets with the use of [..] instead of {..}.

It is not hard to see that the formal semantics captured by these definitions exactly correspond to the intuitive semantics discussed earlier for Queries $Q1$ and $Q2$ of Section 4.2.

4.3.3 Aggregation with Fixed Output Schema

In SQL, we are restricted to "vertical" (or column-wise) aggregation on a *predetermined* set of columns, while *SchemaSQL* allows "horizontal" (or row-wise) aggregation, and also aggregation over more general "blocks" of information. Before we illustrate these points with examples, we provide a formal development of the semantics.

Semantics of Aggregation with Fixed Output Schema

The development of the semantics of aggregation in *SchemaSQL* queries with a fixed output schema is similar to that for SQL aggregation. However, there is a great difference in the expressive power: in SQL, we are restricted to "vertical" (or column-wise) aggregation on a predetermined set of columns, while *SchemaSQL* allows "horizontal" (or row-wise) aggregation, and also aggregation over more general "blocks" of information. Examples of aggregation in *SchemaSQL* were given in section 4.3.3. Here we formalize the semantics.

Let Q be a *SchemaSQL* query involving aggregation. We define an equivalence relation $\sim$ on the instantiations $\mathcal{I}_Q$ as follows.

Definition 4.3.4 *For* $\imath, \jmath \in \mathcal{I}_Q$, $\imath \sim \jmath$ *iff* $\forall$"v" $\in$ groupbyList, $\imath(v) = \jmath(v)$. *It is straightforward to see that* $\sim$ *is an equivalence relation on* $\mathcal{I}_Q$. *Intuitively, two instantiations are* $\sim$*-equivalent provided they agree on all variables appearing in the* group by *clause.*

Let $\mathcal{T}_{\texttt{aggList}}$ denote the set of tuples over the scheme aggList. We define a function $aggregate_Q : \mathcal{I}_Q \rightarrow \mathcal{T}_{\texttt{aggList}}$ as follows.

$$aggregate_Q(\imath) = \bigotimes_{\text{"}agg_v(v)\text{"} \in \texttt{aggList}} agg_v([\jmath(v) \mid \jmath \in \mathcal{I}_Q, \jmath \sim \imath])$$

For a given instantiation $\imath$, $aggregate_Q$ performs the operation agg_v indicated on the variable v in the aggList on the *multiset* of values associated with this variable by all instantiations equivalent to $\imath$.

Now, we are ready to describe the tuple assembly associated with aggregate queries. Let Q be a query involving aggregation. We define a function $aggtuple_Q : \mathcal{I} \rightarrow \mathcal{T}_{\texttt{itemList}} \times \mathcal{T}_{\texttt{aggList}}$ as follows.

$$aggtuple_Q(\imath) = tuple_Q(\imath) \bigotimes aggregate_Q(\imath)$$

Finally, the result of an aggregate query is captured by the function

$Q(D) = \{aggtuple_Q(\imath) \mid \imath \in \mathcal{I}_Q$ & the tuple $aggtuple_Q(\imath)$ satisfies the conditions haveConditions in the having clause$\}$.

We now provide some examples illustrating aggregation in *SchemaSQL* .

Example 4.3.1 The query

```
      select   T.category, avg(T.D)
(Q3)  from     univ-B::salInfo-> D,
               univ-B::salInfo T
      where    D <> "category"
      group by T.category
```

computes the average salary floor of each category of employees over *all* departments in univ-B. This captures horizontal aggregation. The condition D <> "category" enforces the variable D to range over department names. Hence a knowledge of department names (and even the number of departments) is not required to express this query. Alternatively, we could enumerate the departments, e.g., use the condition (D = "Math" or D = "CS" or ...)[5]. By contrast, the query

```
      select T.category, avg(T.salFloor)
(Q4)  from univ-C-> D, univ-C::D T
      group by T.category
```

computes a similar information from univ-C. Notice that the aggregation is computed over a multiset of values obtained from *several relations* in univ-C. In a similar way, aggregations over values collected from more than one database can also be expressed. Block aggregations of a more sophisticated form are illustrated in Example 4.4.3. ■

4.4 Semantics II: Dynamic Output Schema and Restructuring Views

The result of an SQL query (or view definition) is a single relation. Our discussion in the previous section was limited to the fragment of *SchemaSQL* queries that produce one relation, with a fixed schema, as output. In this section, we provide examples to demonstrate the following capabilities of *SchemaSQL* . (i) *declaration of dynamic output schema*, (ii) *restructuring views*, and (iii) *interaction between dynamic output schema creation and aggregation.*

[5]An elegant solution would be to specify some kind of "type hierarchy" for the attributes which can then be used for saying "D is an attribute of the following *kind*", rather than "D is one of the following attributes". Our proposed extension to *SchemaSQL* discussed in Section 4.6, addresses this issue.

We illustrate the capabilities of *SchemaSQL* for the generation of an output schema which can dynamically depend on the input instance (*i.e.* the databases in the federation). While aggregation in SQL is restricted to vertical aggregation on a predetermined set of columns, we have so far seen that *SchemaSQL* can express horizontal aggregation and aggregation over more general "blocks" (see Example 4.3.1). In this section, we shall see that the combination of dynamic output schema and meta-data variables allows us to express vertical aggregation on a *variable number of columns* as well.

4.4.1 Restructuring Without Aggregation

We first illustrate the ideas and expressive power of *SchemaSQL* for performing restructuring, using examples. Formal development will follow.

Example 4.4.1 Consider the relation salInfo in the database univ-B. The following *SchemaSQL* view definition restructures this information into the format of the schema univ-A::salInfo.

```
create view
BtoA::salInfo(category, dept, salFloor) as
        select      T.category, D, T.D
(Q5)    from        univ-B::salInfo-> D,
                    univ-B::salInfo T
        where       D <> 'category'
```

Explanation: Two variables are declared in the from clause: T is a tuple variable ranging over the tuples of relation univ-B::salInfo, and D is an attribute-name variable ranging over the attributes of univ-B::salInfo. The condition in the where clause forces D to be a department name. Finally, each output tuple (T.category,D,T.D) lists the category, department name, and the corresponding salary floor (which is in the format of univ-A::salInfo).

Note that each tuple in the univ-B::salInfo format generates several tuples in the univ-A::salInfo scheme. The mapping, in this respect, is one-to-many. But each *instantiation* of the variables in the query, actually contributes to one output tuple. ■

The following example illustrates restructuring involving dynamic creation of output schema.

Example 4.4.2 This view definition restructures data in `univ-A::salInfo` into the format of the schema `univ-B::salInfo`.

```
      create view AtoB::salInfo(category, D) as
       select   A.category, A.salFloor
(Q6)   from     univ-A::salInfo A, A.dept D
```

Explanation: Each tuple of `univ-A::salInfo` contains the salary floor for one category in a single department, while each tuple of `univ-B::salInfo` contains the salary floors for one category in every department. Intuitively, all tuples in `univ-A::salInfo` corresponding to the same category are grouped together and "merged" to produce one output tuple.

Another aspect of this restructuring view is the use of variables in the `create view` clause. The variable `D` in `create view AtoB::salInfo(category, D)` is declared as a domain variable ranging over the values of the `dept` attribute in the relation `univ-A::salInfo`. Hence, the schema of the view `AtoB::salInfo` is "dynamically" declared as `AtoB::salInfo(category, dept1, ..., deptn)`, where `dept1, ..., deptn` are the values occurring in the `dept` column in the relation `univ-A::salInfo`.

The restructuring in this example corresponds to a many-to-one mapping from instantiations to output tuples. ■

As demonstrated by the previous examples, the semantics of restructuring in the context of a dynamically declared output schema has two aspects to it: `(i)` the determination of the output schema itself, and `(ii)` the formatting of data to conform to the schema determined in `(i)`. In this section, we formalize these concepts. We illustrate our development of the semantics by revisiting Example 4.4.2.

Let a query Q be a view definition of the form

```
create view db::rel(attr1, ..., attrn) as
select  obj1, ..., objn
from    fromList
where   whereConditions
```

We first define some useful notions. Recall that $\mathcal{I}_Q$ is the set of instantiations satisfying the conditions in the `where` clause, as defined in Section 4.3.2.

Definition 4.4.1 *For two instantiations $\imath, \jmath \in \mathcal{I}_Q$, we define $\imath \equiv \jmath$, provided $\imath(\mathtt{db}) = \jmath(\mathtt{db})$ and $\imath(\mathtt{rel}) = \jmath(\mathtt{rel})$. Clearly $\equiv$ is an equivalence relation.*

Determination of output schema: Each instantiation $\imath \in \mathcal{I}_Q$ produces a view of a database, $\imath(\texttt{db})$, containing a relation named $\imath(\texttt{rel})$, whose scheme consists of the attribute set $attrset_Q(\imath) = \{\jmath(\texttt{attr}) \mid \texttt{attr} \in \{\texttt{attr1}, ..., \texttt{attrn}\},\ \jmath \in \mathcal{I}_Q,\ \jmath \equiv \imath\}$. Thus, each $\equiv$-equivalence class of instantiations defines one relation scheme in the output view. *E.g.*, in Example 4.4.2, all instantiations $\imath \in \mathcal{I}_Q$ are $\equiv$-equivalent, and this one equivalence class produces a view containing a database called AtoB, containing one relation named salInfo, with the scheme {category, CS, Math, ...}.

Formatting data to fit the schema: There are two aspects to this. Firstly, the output computed by the *SchemaSQL* query defining the view has to be properly *allocated* to conform to the output schema declared in the **create view** statement. This by itself might in general result in null values, which are eliminated by identifying maximal subsets of "related" tuples and "merging" them. These ideas are made precise below.

As seen above, an instantiation $\imath \in \mathcal{I}_Q$ contributes to a view of a database $\imath(\texttt{db})$ containing a relation named $\imath(\texttt{rel})$ with a scheme given by $attrset_Q(\imath)$. The instantiations $\equiv$-equivalent to $\imath$ contribute to a relation, $allocate_Q(\imath)$, over the attribute set $attrset_Q(\imath)$, as follows. For each instantiation $\jmath \in \mathcal{I}_Q$ such that $\jmath \equiv \imath$, $allocate_Q(\imath)$ contains a tuple t, defined as follows. Let $A \in attrset_Q(\imath)$. Then

$$t[A] = \begin{cases} \jmath(\texttt{objk}), & \text{whenever } A = \jmath(\texttt{attrk}) \\ null, & \text{otherwise.} \end{cases}$$

Figure 9(i) shows $allocate_{Q6}(\imath)$ for the view (Q6) defined in Example 4.4.2, where $\imath$ is any instantiation (recall all of them are $\equiv$-equivalent).

AtoB

salInfo

category	CS	Math
Prof	65,000	null
Assoc Prof	50,000	null
Prof	null	60,000
Assoc Prof	null	55,000

(i)

univ-B

salInfo

category	CS	Math
Prof	65,000	60,000
Assoc Prof	50,000	55,000

(ii)

Figure 9: (i) The relation $allocate_{Q6}(\imath)$ and (ii) the final result after merging

Secondly, merging of tuples in $allocate_Q(\iota)$ is formalized as follows. Let DOM denote the union of all domains of all attributes of all relations involved in the federation, together with the null value, *null*. Define a partial order on DOM, by setting $null \leq v$, $\forall v \in$ DOM. In particular, note that any two distinct non-null values are incomparable. The *least upper bound*, *lub*, of two values in DOM is defined in the obvious way.

$$lub(u, v) = \begin{cases} u, & \text{if } v \leq u \\ v, & \text{if } u \leq v \\ undefined, & \text{otherwise.} \end{cases}$$

We now have the following

Definition 4.4.2 *Two tuples t_1, t_2 over a relation scheme $R = \{A_1, \ldots, A_n\}$ are* mergeable *provided for each $i = 1, \ldots, n$, either $t_1[A_i] = t_2[A_i]$, or at least one of $t_1[A_i]$ or $t_2[A_i]$ is a null. Suppose t_1 and t_2 are mergeable. Then their* merge, *denoted $t = t_1 \odot t_2$, is defined as $t[A_i] = lub(t_1[A_i], t_2[A_i])$, $i = 1, \ldots, n$.*

Clearly, the operator $\odot$ is commutative and associative, and it can be easily extended to any set of *mergeable* tuples. It will be convenient below to extend the operator $\odot$ to any relation containing an arbitrary (*i.e.* *not* necessarily mergeable) set of tuples. The idea is to merge every maximal subset of mergeable tuples in the relation.

Definition 4.4.3 *Let r be any relation over the scheme R. Then the* merge *of r, denoted $\odot r$, is defined as follows.*

$\odot r = \{t \mid \exists a \textit{ maximal subset} \{t_1, \ldots, t_m\} \subseteq r$, *of mergeable tuples, and* $t = \odot\{t_1, \ldots, t_m\}\}$.

Note that $\odot\{t_1, \ldots, t_m\}$ is as defined in Definition 4.4.2.

Finally, we can define the semantics of view definitions in *SchemaSQL* as follows. *The output relation produced by instantiations $\equiv$-equivalent to $\iota \in \mathcal{I}_Q$ is given by $\odot allocate_Q(\iota)$.* *E.g.*, the final output produced by the view definition (Q6) in Example 4.4.2 is a view of a database A2B containing a relation `salInfo(category, CS, Math)` as shown in Figure 9(ii).

4.4.2 Aggregation with Dynamic View Definition

In Section 4.3, we illustrated the capability of *SchemaSQL* for computing (i) horizontal aggregation and (ii) aggregation over blocks of information collected from several relations, or even databases. In this section, we shall see that when *SchemaSQL* aggregation is combined with its view definition facility, it is possible to express vertical aggregation over a *variable* number of columns, determined dynamically by the input instance. The following example illustrates this point.

Example 4.4.3 Suppose that in the database univ-D in Figure 8, there is an additional relation faculty(dname, fname) relating each department to its faculty. Consider the query

```
       select U.fname, avg(T.C)
       from univ-D::salInfo-> C,
(Q7)   univ-D::salInfo T, univ-D::faculty U
       where C <> "dept" and T.dept = U.dname
       group by U.fname
```

Q7 computes, for each faculty, the faculty-wide average floor salary of *all* employees (over all departments) in the faculty. Notice that the aggregation is performed over 'rectangular blocks' of information. Consider now the following view definition Q8, which is essentially defined using the query Q7.

```
    create view averages::salInfo(faculty, C) as
       select U.fname, avg(T.C)
       from univ-D::salInfo-> C,
(Q8)   univ-D::salInfo T, univ-D::faculty U
       where C <> "dept" and T.dept = U.dname
       group by U.fname
```

The view defined by Q8 actually computes, for each faculty, the average floor salary in *each category* of employees (over all departments) in the faculty. This is achieved by using the variable C, ranging over categories, in the dynamic output schema declaration through the create view statement. ∎

The semantics of restructuring (via view definition) with aggregation involves putting together the ideas behind each of these operations. Intuitively, as explained in Section 4.4.1, the instantiations $\equiv$-equivalent to $\imath \in \mathcal{I}_Q$ produce (in the view) one relation in a database whose scheme consists of the attributes $attrset_Q(\imath)$, as defined in that section. The tuples for this relation are obtained by computing the aggregations listed in the aggList in the select statement w.r.t. each equivalence class of instantiations which agree on the variables listed in the group by clause *as well as* on the variables appearing in the create view statement, and then performing the necessary merging. This intuition is formalized below. Consider the view definition Q below where we omit the having clause for simplicity.

```
create view db::rel(attr1, ..., attrn) as
select objList, aggList
from fromList
where whereConditions
group by groupbyList
```

Definition 4.4.4 *For two instantiations* $\imath, \jmath \in \mathcal{I}_Q$, *we define* $\imath \# \jmath$, *provided for each* $o \in$ groubyList, $\imath(o) = \jmath(o)$, *and for each variable* X *occurring in the* create view *statement*, $\imath(X) = \jmath(X)$. *Clearly*, $\#$ *is an equivalence relation.*

The notions of $aggregate_Q(\imath)$ and $aggtuple_Q(\imath)$ are defined analogously to the way they were defined in Section 4.4.1. The only difference is that the equivalence relation $\#$ is used instead of $\sim$.

$$\begin{aligned} aggregate_Q(\imath) &= \bigotimes_{\text{“}agg_o(o)\text{”} \in aggList} agg_o([\jmath(o) \mid \jmath \in \mathcal{I}_Q, \jmath \# \imath]) \\ aggtuple_Q(\imath) &= tuple_Q(\imath) \bigotimes aggregate_Q(\imath) \end{aligned}$$

The concept of *allocating* tuples computed above according to the various output schemas dynamically created by the instantiations can be formalized in a way similar to what was done in Section 4.4.1, and we suppress these obvious details for brevity. Let $aggallocate_Q\ (\imath)$ denote the allocated relation determined by the $\equiv$-equivalence class of $\imath \in \mathcal{I}_Q$. The concept of merging, defined in Definitions 4.4.2 and 4.4.3, can now be directly applied to compute the final output. Thus, for each instantiation $\imath \in \mathcal{I}_Q$, the $\equiv$-equivalence class of $\imath$ contributes to the relation $\bigodot aggallocate_Q\ (\imath)$, over the schema $\imath(\texttt{db}) :: \imath(\texttt{rel})(attrset_Q(\imath))$.

E.g., it is easy to verify that the view defined by Q10 indeed computes for each faculty, the category-wise floor salary averages. Before closing this section, we note that the combination of dynamic output schema declaration with *SchemaSQL* ś aggregation mechanism makes it possible to express many other novel forms of aggregation as well.

4.5 Implementation Issues

In this section we describe the architecture of a system for implementing a multidatabase querying and restructuring facility based on *SchemaSQL*. A highlight of our architecture is that it builds on existing architecture in a *non-intrusive way*, requiring minimal extensions to prevailing database technology. This makes it possible to build a *SchemaSQL* system on top of (already available) SQL systems. We also identify novel query optimization opportunities that arise in a multidatabase setting.

The architecture consists of a *SchemaSQL* server that communicates with the local databases in the federation. We assume that the meta-information comprising of component database names, names of the relations in each database, names of the attributes in each relation, and possibly other useful information (such as statistical information on the component databases useful for query optimization) are stored in the *SchemaSQL* server in the form of a relation called *Federation System Table* (FST). Due to the varying degrees of autonomy component databases enjoy in a multidatabase system, some or all of this information may not be available. While our approach is capable of handling this scenario, for the sake of clarity, we assume that the component database names as well as their schema information is available in the *SchemaSQL* server.

In our architecture, global *SchemaSQL* queries are submitted to the *SchemaSQL* server, which determines a series of local SQL queries and submits them to the local databases. The *SchemaSQL* server then collects the answers from local databases, and, using its own *resident* SQL engine, executes a final series of SQL queries to produce the answer to the global query. Intuitively, the task of the *SchemaSQL* server is to compile the *instantiations* for the variables declared in the query, and enforce the conditions, groupings, aggregations, and mergings to produce the output. Many query optimization opportunities at different stages, and at different levels of abstraction, are possible, and should be employed for efficiency (see discussions at the end of this section). Figure 10 depicts our architecture for implementing *SchemaSQL*. Algorithm 4.5.1, gives a more detailed account of our query processing strategy.

Query processing in a *SchemaSQL* environment consists of two major phases. In the first phase, tables called *VIT*'s (Variable Instantiation Table) corresponding to the variable declaration in the **from** clause of a *SchemaSQL* statement are generated. The schema of a VIT consists of all the variables in one or more variable declarations in

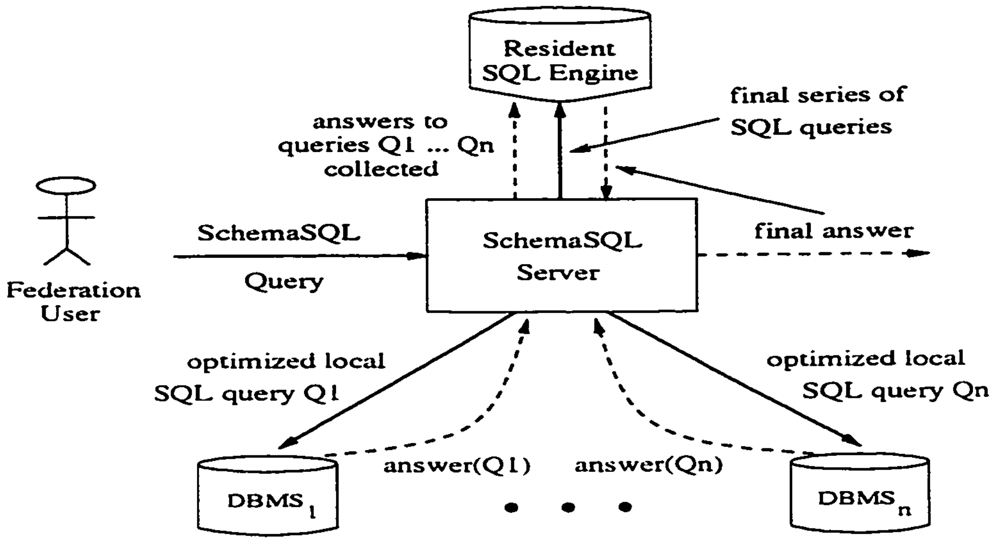

Resident SQL Engine
answers to queries Q1 ... Qn collected
final series of SQL queries
Federation User
SchemaSQL Query
SchemaSQL Server
final answer
optimized local SQL query Q1
optimized local SQL query Qn
answer(Q1)
answer(Qn)
DBMS$_1$
DBMS$_n$

Figure 10: SchemaSQL - Implementation Architecture

the from clause and its contents correspond to instantiations of these variables. *VIT's are materialized by executing appropriate SQL queries on the FST and/or component databases.* In the second phase, the *SchemaSQL* query is rewritten into an equivalent SQL query on the *VIT*'s and the generated answer is appropriately presented to the user. Our algorithm below considers *SchemaSQL* queries with a fixed output schema possibly with aggregation. A complete algorithm for the implementation of the full language, as well as novel query optimization strategies are discussed in [LSS96b].

In the following, we assume that the FST has the scheme FST `(db-name, rel-name, attr-name)`. Also, we refer to the db-name, rel-name, and attr-name variables (defined in Definition 4.2.1) collectively as *meta-variables*.

Algorithm 4.5.1 *SchemaSQL Query Processing*
INPUT: A *SchemaSQL* query with a fixed output schema and aggregation.
OUTPUT: Bindings for the variables appearing in the select clause of the *SchemaSQL* statement.
METHOD: The algorithm consists of two phases.
(1) Corresponding to a set of variable declarations in the from clause, create *VIT*s using one or more SQL queries against some local databases and/or the FST.
(2) Rewrite the original *SchemaSQL* query against the federation into an equivalent query against the set of *VIT* relations and run it using the resident SQL server.
Phase I

(0) The input *SchemaSQL* statement is rewritten into the following form such that the conditions in the **where** clause are in conjunctive normal form.

```
        select $S_1, \ldots, S_n$
        from $\langle range_1 \rangle$ $V_1$, ..., $\langle range_k \rangle$ $V_k$
        where $\langle cond_1 \rangle$ and ... and $\langle cond_m \rangle$
        group by groupList
        having haveConditions
```

(1) Consider the variable declaration for variable V_i.

(a) If V_i is a meta-variable: In this case, all variables in the declaration $\langle range_i \rangle$ V_i range over meta-data. Create VIT_i with a schema consisting of V_i and any variables appearing in $\langle range_i \rangle$, and contents obtained using an appropriate SQL query against the FST. For example, let '`D::rel->` V_i' be the declaration and one of the conditions in the **where** clause be 'V_i *.op.* c' where *.op.* is a (in)equality operator and c is a constant. Obtain VIT_i corresponding to VIT_i as:

```
  select db-name as D, attr-name as $V_i$
from FST
where rel-name = 'rel' and attr-name .op. c
```

Meta-variable declarations of other forms are handled in a similar way.

(b) If V_i is a domain variable: Group together domain variable declarations that are declared using the same tuple variable as for V_i. Create VIT_i with schema consisting of the domain variables in the group. Obtain the (tuple of) bindings for the attr-name variables (in the range declarations) in the group, using their corresponding VIT's. Using this, generate a set of SQL queries against local databases. The contents of VIT_i will be the union of answers to these queries. For example, let db::rel T be a tuple variable declaration, 'T.A V_i' be the declaration for the domain variable and 'V_i *.op.* c' be a condition in the **where** clause, where *.op.* is a (in)equality operator and c is a constant. Let 'T.attr V_j' be another domain variable declaration in the from clause.

(i) Obtain the bindings for attr-name variable A from its VIT, and name it relation T.

(ii) For $a \in T$, generate an SQL query against database db:

```
select a as $V_i$, attr as $V_j$
from rel
where a .op. c
```

(iii) Obtain VIT_i as the union of answers to all the SQL queries generated in (ii), against db.

Domain-variable declarations of other forms (*e.g.* when db, rel, attr are also variables) are handled in a similar way.

(c) If V_i is a tuple variable: Generate bindings for the meta-variables in $\langle range_i \rangle$ as in case (a). The attributes of the VIT corresponding to V_i are obtained by analyzing the *select*, *where*, *group by*, and *having* clauses. We consider a variable V as *relevant* in the context of tuple variable V_i, if (i) V is of the form $V_i.C$ or $V_i.c$ (C, c are a variable and constant respectively) and occurs in the *select*, *group by*, or *having* clause, or (ii) V occurs in the declaration of V_i and either is compared with a variable in the *where* clause, or occurs in the *select* clause, or (iii) V occurs in a relevant variable of the form $V_i.V$ and V is compared with a variable in the *where* clause. The schema of the VIT is the set consisting of all relevant variables in the context of V_i. Finally, the contents of VIT are obtained by generating appropriate SQL queries against local databases. In general, if there are occurrences of the form $V_i.C$ in the *select* or the *where* clause, the VIT would be obtained as a union of several SQL queries.

For example, let the select clause contain an aggregation of the form avg($V_i.C$), the variable declaration be 'db::R V_i' and two of the conditions in the where clause be '$V_i.a_1$ *.op.* $V_j.a_2$' and '$V_i.a_3$ *.op.* c', where a_1, a_2, a_3, c are constants.

(i) Obtain a VIT corresponding to db-> R (as in (a) above) and name it T.

(ii) The schema of VIT_i is $\{V_i.C, V_i.a_1\}$.

(ii) For each $r \in T$, obtain the attribute names in relation r (using an SQL query on the FST) and generate the following SQL statement.
Let $c_1, \ldots, c_k$ be the instantiations of C. corresponding to r.

```
select c_1 as V_i.C, a_1 as V_i.a_1
from r
where V_i.a_3 .op. c
UNION
...
UNION
select c_k as V_i.C, a_1 as V_i.a_1
from r
where V_i.a_3 .op. c
```

(iii) Obtain VIT_i as the union of all the SQL statements generated in (ii).

Tuple variable declarations of other forms are handled in a similar way.

Phase II

Execution of this phase happens in the *SchemaSQL* server. The *SchemaSQL* query is rewritten into an equivalent conventional SQL statement on the VIT's generated in Phase I, in the following way. *(a)* The **select**, **group by**, and **having** clauses of the rewritten query are obtained by copying the corresponding clauses in the *SchemaSQL* query after disambiguating the attribute names that appear in more than one VIT: *(b)* the from clause consists of the subset of VIT's relevant to the final result, and *(c)* the where clause is obtained by retaining the conditions involving tuple variables and by adding a condition '$VIT_i.X = VIT_j.X$' for tables VIT_i and VIT_j having a common attribute.

It is interesting to note that using our algorithm, the novel horizontal aggregation (Section 4.3.3, Example 4.3.1) which cannot be performed in a conventional SQL system, can be easily realized in our framework. More general kind of 'block' aggregations can also be handled in a similar way.

Theorem 4.5.1 *Algorithm 4.5.1 correctly computes answers to* SchemaSQL *queries.*

Proof Sketch:

We prove the correctness of the theorem by establishing that Algorithm 4.5.1 generates the set $Q(D)$, the set of tuples that form the answer to a *SchemaSQL* query Q, according to the semantics of *SchemaSQL* . $Q(D)$ is obtained from I_Q, the set of all 'legal' instantiations for the variables in Q. We show that the various steps in the algorithm correspond to the generation of (elements of) I_Q.

Step 1(a) of Phase I of the algorithm ensures the admissibility (Definition 4.3.2) condition that each meta-variable is instantiated to some items over its appropriate range. 1(b) ensures the validity condition involving domain variables (Definition 4.3.3, condition (b)) 1(c) captures the admissibility condition involving tuple variables. For optimization purposes, some of the **where** conditions are pushed into each of these steps. This corresponds to an early verification of the validity condition, and limits the number of instantiations for the variables in Q. Finally, Phase II corresponds to the tuple assembly, $tuple_Q$ function, and projects the parts of the tuples relevant to output. Note that the condition $sat(I, Q)$ is identical for the semantics of SQL as well as that of *SchemaSQL* (with respect to the instantiations) and hence the conditions in the **where** clause, not used in Phase I are retained. A similar argument holds for retaining the aggregation operators as well as the **group by** and **having** clauses.

□

VIT_1

RelC
cs
math

VIT_2

RelC	C.salFloor
cs	50,000
math	40,000

VIT_3

D.dept	D.technician
cs	55,000
math	40,000

Figure 11: Example - Query Processing

Example 4.5.1 In this example, we illustrate our algorithm using a variant of the query $Q2$ of Example 4.2.2.

'List the departments in univ-C that pay a higher salary floor to their technicians compared with the same department in univ-D. List also the (higher) pay.'

```
select   RelC, C.salFloor
from     univ-C-> RelC, univ-C::RelC C,
         univ-D::salInfo D
where    RelC =  D.dept        and
         C.category  = "technician"  and
         C.salFloor  >  D.technician
```

Phase I

VIT_1 corresponding to the variable declaration univ-C-> RelC is created using:

```
select rel-name as RelC
from   FST
where  db-name = 'univ-C'
```

Figure 11 shows VIT_1. To generate the SQL statement that creates VIT_2, the following SQL queries are issued against the FST.

```
select attr-name          select attr-name
from FST                  from FST
where                     where
db-name = 'univ-C'        db-name = 'univ-C'
and rel-name = 'cs'       and rel-name = 'math'
```

Let the answer to both the queries be {*category*, *salFloor*}. VIT_2, corresponding to `univ-C::RelC C` is obtained by querying the database univ-C using:

```
select 'cs' as RelC, cs.salFloor as C.salFloor
from    cs
where   cs.category = 'technician'
        UNION
select 'math' as RelC,
        math.salFloor as C.salFloor
from    math
where   math.category = 'technician'
```

To obtain VIT_3 corresponding to `univ-D::salInfo D`, querying is first done on the FST to obtain the names of the attributes in relation *salInfo* of database univ-D:

```
select attr-name
from   FST
where db-name = 'univ-D' & rel-name = 'salInfo'
```

Let the answer to this query be {*dept*, *prof*, *technician*}. VIT_3, shown in Figure 11 is obtained by querying the database univ-D:

```
select  dept as D.dept,
        technician as D.technician
from    salInfo
```

Phase II

Having obtained all the VIT's corresponding to the variable declarations, Phase II now consists of rewriting the *SchemaSQL* statement into the following SQL statement to obtain the final answer.

```
select RelC, C.salFloor
from      $VIT_2, VIT_3$
where RelC = D.dept and
            C.salFloor > D.technician
```

■

A *SchemaSQL* system on the PC-Windows platform is currently under implementation.

Query Optimization

There are several opportunities for query optimization which are peculiar to the MDBS environment. In the following, we identify the major optimization possibilities and sketch how they can be incorporated in Algorithm 4.5.1.

1. The conditions in the **where** clause of the input *SchemaSQL* query should be pushed inside the local spawned SQL queries so that they are as 'tight' as possible. Algorithm 4.5.1 incorporates this optimization to some extent.
2. Knowledge of the variables in the **select** and **where** clauses can be used to minimize the size of the VIT's generated in Phase I. For example, if certain attributes are not required for processing in Phase II, they can 'dropped' while generating the local SQL queries.
3. If more than one tuple variable refers to the same database, and their relevant where conditions do not involve data from another database, the SQL statements corresponding to these variable declarations should be combined into one. This would have the effect of combining the VIT's corresponding to these variable declarations and thus reducing the number of spawned local SQL queries. This can be incorporated by modifying the step 1(c) of our algorithm.
4. One of the costliest factors for query evaluation in a multidatabase environment is database connectivity. We should minimize the number of times connections are made to a database during query evaluation. Thus, the spawned SQL statements need to be submitted (in batches) to the component databases in such a way that they are evaluated in minimal number of connections to the databases.
5. In view of the *sideways information passing (sip)* [BR86] technique inherent in our algorithm, reordering of variable declarations would result in more efficient query processing. However, the heuristics that meta-variables obtain

a significantly fewer number of bindings when compared to other variables in a multidatabase setting, presents novel issues in reordering. For instance the order `'db::r.a R, -> D, D-> R'` suggested by the conventional reordering strategies could be worse than `'-> D, D-> R, db::r.a R'` because of the lower number of bindings R obtains for VIT_2 in the latter.

6. We should make use of works such as [LN90, LNS90] to determine which of the VIT's should be generated first so that the tightest bindings are passed for generating subsequent VITs.
7. If parallelism can be supported, SQL queries to multiple databases can be submitted in parallel.

Replication and Inconsistency

Replication of data, and inconsistency among data from local databases are common in multidatabase systems. The view facility of *SchemaSQL* and our architecture provide the means to cope with these difficulties.

Controlled (intentional) replication can be addressed through the Federation System Table, FST. A copy of the replicated data is identified as the *primary copy*, and the FST routes all references to the replicated data to the primary copy. The choice of the primary copy is influenced by factors such as efficiency of query processing, network connectivity, and the load at local sites. In a dynamic scheme, the FST is updated in response to changes in the network (e.g., network disconnection) and the load at local sites.

Data replication and overlap among (independent) local sites, with the possibility of inconsistency, is much subtler. The view facility of *SchemaSQL* can be used to resolve inconsistencies by exposing only the appropriate data through the view. This is similar to the approach taken in multidatabase systems utilizing an (integrated) global schema. Our architecture is more flexible, and does not require a global schema, yet, the view facility can mimic the role played by the global schema for resolving data inconsistency.

4.6 Semantic Heterogeneity

One of the roadblocks to achieving true interoperability is the heterogeneity that arises due to the difference in the meaning and interpretation of similar data across the component systems. This *semantic heterogeneity* problem has been discussed in detail in [Sig91], [KCGS93], [HM93]. A promising approach to dealing with semantic heterogeneity is the proposal of Sciore, Siegel, and Rosenthal [SSR94]. The main idea behind their proposal is the notion of *semantic values*, obtained by introducing explicit context information to each data object in the database. In applying this idea to the relational model, they develop an extension of SQL called Context-SQL (C-SQL) that allows for explicitly accessing the data as well as its context information.

In this section, we sketch how *SchemaSQL* can be extended with the wherewithal to tackle the semantic heterogeneity problem. We extend the proposal of [SSR94], by associating the context information to relation names as well as attribute names, in addition to the values in a database. Also, in the *SchemaSQL* setting, there is a natural need for including the type information of an object as part of its context information. We propose techniques for intensionally specifying the semantic values as well as for algorithmically deriving the (intensional) semantic value specification of a restructured database, given the old specification and the *SchemaSQL* view definition. The following example illustrates our ideas.

Example 4.6.1 Consider the database univInfoA having a single relation **stats** with scheme {**cat**, cs, **math**, **ontario**, **quebec**}. This database stores information on the floor salary of various employee categories for each department (as in univ-B of the university federation) as well as information on the average number of years it takes to get promoted to a category, in each province in the country. The type information of the objects in the database univInfoA is stored in a relation called *isa* and is captured using the following rules[6]:

$isa(cs, dept) \leftarrow$

$isa(math, dept) \leftarrow$

$isa(ontario, prov) \leftarrow$

$isa(quebec, prov) \leftarrow$

$isa(C, cat) \leftarrow stats[cat \rightarrow C]$

$isa(S, sal) \leftarrow stats[D \rightarrow S], isa(D, dept)$

$isa(Y, year) \leftarrow stats[P \rightarrow Y], isa(P, prov)$

Now, consider restructuring univInfoA into univInfoB which consists of two relations **salstats{dept, prof, assoc-prof}** and **timestats{prov, prof, assoc-prof}**. **salstats** has tuples

[6]The syntax of the type specification rules is based on the syntax of *SchemaLog* [LSS97].

of the form $< d, s_1, s_2 >$. representing the fact that d is a department that has a floor salary of s_1 for category professor, and s_2 for associate professor. A tuple of the form $< p, y_1, y_2 >$ in **timestats** says that p is a province in which the average time it takes to reach the category professor is y_1 and to reach the category associate professor is y_2. The following *SchemaSQL* statements perform the restructuring that yields univInfoB.

```
create view
       univInfoB::salstats(dept, T.cat) as
select D, T.D
from   univInfoA::stats T,
       univInfoA::stats-> D,
where  D isa 'dept'

create view
       univInfoB::timestats(prov, T.cat) as
select P, T.P
from   univInfoA::stats T,
       univInfoA::stats-> P,
where  P isa 'prov'
```

Note how the type information is used in the **where** clause to elegantly specify the range of the attribute variables. Our algorithm that processes the restructuring view definitions derives the following intensional type specification for univInfoB:

$isa(prof, cat) \leftarrow$

$isa(assoc\text{-}prof, cat) \leftarrow$

$isa(D, dept) \leftarrow salstats[dept \rightarrow D]$

$isa(S, sal) \leftarrow salstats[C \rightarrow S], isa(C, cat)$

$isa(P, prov) \leftarrow timestats[prov \rightarrow P]$

$isa(Y, year) \leftarrow timestats[P \rightarrow Y], isa(P, prov)$

Query processing in this setting involves the following modification to the processing of comparisons mentioned in the user's query. The comparison is performed after (a) finding the type information using the specification, (b) finding the associated context information, and (c) applying the appropriate conversion functions.

4.7 Comparison with Related Work

In this section, we compare and contrast our proposal against some of the related work for meta-data manipulation and multidatabase interoperability.

The features of *SchemaSQL* that distinguishes it from similar works include

- Uniform treatment of data and metadata.
- No explicit use of object identifiers.
- Downward compatibility with SQL.
- Comprehensive aggregation facility.
- Restructuring views, in which data and meta-data may be interchanged.
- Designed specifically for interoperability in multi-database systems.

Further, we also discuss the implementation of *SchemaSQL* on a platform of SQL servers.

In [Lit89, GLRS93], Litwin *et al.* propose a multidatabase manipulation language called MSQL that is capable of expressing queries over multiple databases in a single statement. MSQL extends the traditional functions of SQL to the context of a federation of databases. The salient features of this language include the ability to retrieve and update relations in different databases, define multi-database views, and specify compatible and equivalent domains across different databases. [MR95] extends MSQL with features for accessing external functions (for resolving semantic heterogeneity) and for specifying a global schema against which the component databases could be mapped. Though MSQL (and its extension) has facilities for ranging variables over multiple database names, its treatment of data and meta-data is non-uniform in that relation names and attribute names are not given the same status as the data values. The issues of schema independent querying and resolving schematic discrepancies of the kind discussed in this chapter, are not addressed in their work.

Many object-oriented query languages, by virtue of treating the schema information as objects, are capable of powerful meta-data querying and manipulation. Some of these languages include XSQL (Kifer, Kim, and Sagiv [KKS92]), HOSQL (Ahmed *et al.* [ASD$^+$91]), Noodle (Mumick and Ross [MR93]), and OSQL (Chomicki and Litwin [CL93]).

XSQL ([KKS92]) has its logical foundations in F-logic ([KLW95]) and is capable of querying and restructuring object-oriented databases. However, it is not suitable for the needs addressed in this chapter as its syntax was not designed with interoperability as a main goal. Besides, the complex nature of this query language raises concerns about effective and efficient implementability, a concern not addressed in [KKS92]. The Pegasus Multi-database system ([ASD+91]) uses a language called HOSQL as its data manipulation language. HOSQL is a functional object-oriented language that incorporates non-procedural statements to manipulate multiple databases. OSQL ([CL93]), an extension of HOSQL is capable of tackling schematic discrepancies among heterogeneous object-oriented databases with a common data model. Both HOSQL and OSQL do not provide for ad-hoc queries that refer to many local databases in the federation in one shot. While XSQL, HOSQL, and OSQL have a SQL flavor, unlike *SchemaSQL* , they do not appear to be downward compatible with SQL syntax and semantics. In other related work, [Ros92] proposes an interesting algebra and calculus that treats relation names at par with the values in a relation. However, its expressive power is limited in that attribute names, database names, and comprehensive aggregation capabilities are not supported.

In [LBT92], Lefebvre, Bernus, and Topor use F-logic ([KLW95]), to reconcile schematic discrepancies in a federation of relational databases. Unlike *SchemaSQL* which can provide a 'dynamic global schema', ad hoc queries that refer the data and schema components of the local databases in a single statement cannot be posed in their framework.

UniSQL/M [KGK+95] is a multidatabase system for managing a heterogeneous collection of relational database systems. The language of UniSQL/M, known as SQL/M, provides facilities for defining a global schema over related entities in different local databases, and to deal with semantic heterogeneity issues such as scaling and unit transformation. However, it does not have facilities for manipulating metadata. Hence features such as restructuring views that transform data into metadata and vice versa, dynamic schema definitions, and extended aggregation facilities supported in *SchemaSQL* are not available in SQL/M. The emerging standard for SQL3 ([SQL96, Bee93]) supports ADTs and oid's, and thus shares some features with higher-order languages. However, even though it is computationally complete, to our knowledge it does not *directly* support the kind of higher-order features in *SchemaSQL*.

Krishnamurthy and Naqvi [KN88] and Krishnamurthy, Litwin, and Kent [KLK91] are early and influential proposals that demonstrated the power of using variables that uniformly range over data and meta-data, for schema browsing and interoperability.

While such 'higher-order variables' admitted in *SchemaSQL* have been inspired by these proposals, there are major differences that distinguish our work from the above proposals. (i) These languages have a syntax closer to that of logic programming languages, and far from that of SQL. (ii) More importantly, these languages do not admit tuple variables of the kind permitted in *SchemaSQL* (and even SQL). This limits their expressive power. (iii) Lastly, aggregate computations of the kind discussed in Sections 4.3.3 and 4.4.2 are unique to our framework, and to our knowledge, not addressed elsewhere in the literature.

In the context of multi-dimensional databases (MDDB) and on-line analytical processing (OLAP), there is a great need for powerful languages expressing complex forms of aggregation ([CCS95]). The powerful features of *SchemaSQL* for horizontal and block aggregation will be especially useful in this context (*e.g.* see Examples 4.3.1, 4.4.3). We have recently observed that the *Data Cube* operator proposed by Gray et al. ([GBLP96]) can be simulated in *SchemaSQL*. Unlike the cube operator, *SchemaSQL* can express any subset of the data cube to any level of granularity.

In other related work, Gyssens et al. ([GLS96]) develop a general data model called the *Tabular Data Model*, which subsumes relations and spreadsheets as special cases. They develop an algebra for querying and restructuring tabular information and show that the algebra is complete for a broad class of natural transformations. They also demonstrate that the tabular algebra can serve as a foundation for OLAP. Restructuring views expressible in *SchemaSQL* can also be expressed in their algebra but they do not address aggregate computations.

We now compare *SchemaSQL* with SchemaLog, discussed in Chapter 2. *SchemaSQL* has been to a large extent inspired by SchemaLog. Indeed, the logical underpinnings of *SchemaSQL* can be found in SchemaLog. However, *SchemaSQL* is *not* obtained by simply "SQL-izing" SchemaLog. There are important differences between the two languages. (i) *SchemaSQL* has been designed to be as close as possible to SQL. In this vein, we have developed the syntax and semantics of *SchemaSQL* by extending that of SQL. SchemaLog on the other hand has a syntax based on logic programming. (ii) Answers to *SchemaSQL* queries come with an associated schema. In SchemaLog, as in other logic programming systems, answers to queries are simply a set of (tuples of) bindings of variables in the query (unless explicitly specified using a restructuring rule). (iii) The aggregation semantics of *SchemaSQL* is based on a 'merging' operator. There is no obvious way to simulate merging in SchemaLog.

(iv) To facilitate an ordinary SQL user to adapt to *SchemaSQL* in an easy way, we have designed *SchemaSQL* without the following features present in SchemaLog - *(a)* function symbols and *(b)* explicit access to tuple-id's. As demonstrated in this chapter, the resulting language is simple, yet powerful for the interoperability needs in a federation.

4.8 Conclusions

In this chapter, we discussed a principled extension of SQL, called *SchemaSQL* , that offers the capability of uniform manipulation of data and meta-data in relational multi-database systems. We developed a precise syntax and semantics of *SchemaSQL* in a manner that extends traditional SQL syntax and semantics, and demonstrated the following. (1) *SchemaSQL* retains the flavour of SQL while supporting querying of both data and meta-data. (2) It can be used to represent data in a database in a structure substantially different from the original database, in which data and meta-data may be interchanged. (3) It also permits the creation of views whose schema is dynamically dependent on the contents of the input instance. (4) While aggregation in SQL is restricted to values occurring in one column at a time, *SchemaSQL* permits "horizontal" aggregation and even aggregation over more general "blocks" of information. (5) *SchemaSQL* provides a great facility for interoperability and data/meta-data management in relational multi-database systems. We provided many examples to illustrate our claims. We outlined an architecture for the implementation of *SchemaSQL* and discussed implementation algorithms based on available database technology that allows for powerful integration of SQL based relational DBMS.

The work presented in this chapter has opened up avenues for interesting research in the context of query languages for OLAP-based information systems. As pointed out in Section 4.7, the aggregation and restructuring features of *SchemaSQL* smack of similar functionalities required in the OLAP setting. However, in order to serve as a full-fledged OLAP query language, *SchemaSQL* needs to be extended further. [Gin97] addresses these and related issues.

Chapter 5

Extending SchemaLog to Legacy and Non-traditional Database Systems

While we proposed SchemaLog as a logic born out of pragmatic needs arising in the context of relational databases, the power and applicability of SchemaLog are by no means confined to the relational model. In this chapter, we study how the thoughts that went into the design of SchemaLog can be brought to bear in a natural way, in the context of legacy databases (such as Network databases) and non-traditional information systems (such as the World Wide Web). We show that the powerful features of SchemaLog for meta-data querying, information restructuring, and interoperability are versatile as well, and extend naturally to settings radically different from the relational model.

We illustrate the natural applicability SchemaLog exhibits for catering to the legacy and non-traditional database systems by studying how the syntax of SchemaLog (perhaps with minor modifications) can be interpreted against the ER databases, CODASYL (Network) databases, and the information repositories on the World Wide Web. These specific information sources we consider, form a representative sample of the many existing non-relational sources and help illustrate the generality of a SchemaLog-based approach for interoperability. Given the importance of the Web, we give it a special treatment in this chapter, and study in depth how it can benefit from a SchemaLog-based approach for querying and restructuring information.

5.1 SchemaLog and the Network Model

The network model is one of the early successful models of databases. It was adopted in the first database-standard specification, called the CODASYL DBTG report, written in the late 1960s by the Database Task Group. We do not discuss the details of the network model here and refer to [Ull84] for a comprehensive treatment of the subject.

The basic concepts in the network model are *record sets* that represent data and *link sets* that capture the relationships among the data. The CODASYL standard allows only for many-to-one links (with one-to-one link as a special case). Whenever there is a link (many-to-one mapping) from one record set to another, we can associate a subset of records of the former (called *member* of the link occurrence) to a single record of the latter (called *owner of the link occurrence*). Thus, the link set can be viewed as a restricted (binary) relationship in the sense of the ER-model. Indeed, a network model schema of a database can be derived from its ER-model representation [Ull84, WK79].

5.1.1 Interpreting SchemaLog Against Network Databases

The intuition behind our approach to adapting the SchemaLog syntax to the network model is based on the principles underlying the methodology for translating a ER-model representation to a network schema. We observe the following about the network schema: the logical record sets are similar to the entities of the ER-model. The link sets correspond to relationships in the ER-model, and consist of the links (as discussed above) that capture the connection between records of the record set. Thus, the records in a record set can be likened to tuples in tables and the link set can be thought of as a relational representation of the many-to-one mapping the link represents. Each 2-tuple in the latter relation would be a pair of tuple-id's (tid's) representing the relationship that exists between the tuples the tid's stand for. Moreover, we name the columns of this (link) relation *owner* and *member*, our choice of the names inspired by the corresponding terminology in the network model parlance. The table in Figure 5.1.1 sums up this approach.

Note that the above abstraction fits nicely in the framework of SchemaLog due to the fact that the SchemaLog model allows for a first-class treatment of tid's. We use the following example to illustrate these ideas.

NETWORK CONCEPT	OUR ABSTRACTION
Record	Tuple
Record Set R with fields $f_1, \ldots, f_n$	Relation R with attributes $f_1, \ldots, f_n$
Link Set S with owner O and member M	Relation S with attributes 'owner' and 'member', having tuples of the form $\langle t_i, t_j \rangle$ where t_i is a tid of a tuple in relation O and t_j is a tid in relation M such that t_i and t_j stand in relationship S.

Figure 12: Abstracting the Network Model

Example 5.1.1 The application as well as the network schema are from [Ull84, pp. 98]. Consider a company 'abc' that keeps track of information about its customers, their orders, balances, possible suppliers, and the prices for the various items supplied by the supplier. The network schema is given below.

```
RECORD SET suppliers
     1 sname char(20),
     1 saddr char(30);

RECORD SET items
     1 iname char(15)

RECORD SET prices
     1 price real,
     1 iname VIRTUAL
          SOURCE IS items.iname OF OWNER OF itempr
     1 sname VIRTUAL
          SOURCE IS suppliers.sname OF OWNER OF suppr;

RECORD SET persons
     1 name char(20),
     1 addr char(30),
     1 balance real;

RECORD SET orders
```

```
    1 order_no integer,
    1 quantity real;

LINK SET suppr
    OWNER IS suppliers
    MEMBER IS prices;

LINK SET itempr
    OWNER IS items
    MEMBER IS prices;

LINK SET itemord
    OWNER IS items
    MEMBER IS orders;

LINK SET persord
    OWNER IS persons
    MEMBER IS orders;
```

As explained before, we abstract both record sets and link sets as relations. The attributes of the relations corresponding to the former would be the fields of the record set. and the relations corresponding to the latter would have two attributes 'owner' and 'member'. Note that the link relation captures the connection between two tuples having tuple-id's t_i and t_j, by the membership of a tuple $\langle t_i, t_j \rangle$ in it. To illustrate. if *Granola* is an item sold for 10.00 by supplier *ABC Corp.*, the **suppr** relation would contain a tuple $\langle t_1, t_2 \rangle$ where t_1 is the tid corresponding to the tuple in relation **suppliers** that has information on ABC Corp, and t_2 is the tid of the tuple in **prices** that represents the price 10.00. Moreover, the **itempr** relation would contain a tuple $\langle t_2, t_3 \rangle$, where t_3 is the tid of the tuple in relation **items** that has information on Granola.

Q_1 : *List all the suppliers of item Granola and the prices they charge*

The following SchemaLog query against the network database 'abc' expresses this.

$\longleftarrow abc :: items[I : iname \rightarrow `Granola'], abc :: itempr[X : owner \rightarrow I, member \rightarrow J],$
$abc :: prices[J : price \rightarrow P], abc :: suppr[Y : owner \rightarrow K, member \rightarrow J],$
$abc :: suppliers[K : sname \rightarrow S]$

I, J, and K are tid variables that occur in the tid position in the 'record set relations' and in the value position in the 'link set relations'. Using these variables, we navigate the appropriate links of the *abc* database to generate the list of suppliers of Granola and their prices. X and Y are "don't care" existential variables.

Q_2 : *List all the items ordered by Brooks*

The corresponding SchemaLog query is:

$\longleftarrow abc :: persons[I : name \rightarrow `Brooks'], abc :: persord[X : owner \rightarrow I, member \rightarrow J],$
$abc :: itemord[Y : owner \rightarrow K, member \rightarrow J], abc :: items[K : iname \rightarrow P].$

In the following section, we discuss some of the salient features of our approach as well as some of the outstanding issues.

5.1.2 Discussions

The examples of the previous section illustrate how the powerful features of SchemaLog such as the ability to refer to and quantify over tid's in a relation and the place-holder facility for various types of concepts such as relation name and attribute name can be effectively put to use to query network databases. It is interesting to note the following important distinction that occurs in the context of applying SchemaLog for network databases as opposed to any other data sources we have considered so far. In the relational context, for instance, we observed an intricate interplay that occurs between the database names, relation names, attribute names, and values: depending on the context, each of them could take the place of any other. However, tid's have their own distinct status and do not seem to participate in the aforementioned interplay. Interestingly, in the case of the network databases, there is a natural need for the tid's to appear in the value position as well. Indeed, it is this facility that provides the ability to navigate the links of a network database. Our examples amply illustrate how the tid variables appear in the value position and vice versa.

We remark that the above characteristic of the network model, justifies one of our important design decisions behind the design of SchemaLog – that of providing a first class status to all concepts and allowing for their free, unrestricted intermingling. This feature of SchemaLog has been crucial for putting it to use in the context of the network databases.

Implementation Issues

While we have proposed an elegant framework for declaratively *posing* queries (in SchemaLog) against network databases, we have not yet addressed the issues that arise in the context of *executing* these queries against the databases. In the following we briefly sketch two possible implementation strategies.

In the first, and the less favored strategy, the relational abstractions of network databases are *materialized* and the SchemaLog queries are executed against such a (relational) database. An important challenge in this strategy is the realization of the tid's of the link relations. One possible way to implement tid's is by making use of the primary key attributes of the record set relations as a surrogate for the tid's. If efficiency is a consideration, physical addresses (pointers) of the records can be used as the tid's. However, due to the highly dynamic nature of the physical addresses, keeping currency of tid's would be a major challenge.

The second (more preferred) strategy involves *translating* SchemaLog queries into the host language (CODASYL) DML. Investigating such a translation methodology as well as the optimization opportunities it opens up is an interesting research issue that is beyond the scope of this thesis and we plan to pursue it as part of future work.

5.2 Extending SchemaLog to the (Extended) ER Model

Entity-Relationship (ER) model is one of the most important models of information systems. Its ability to capture the semantics of most real-life applications, combined with its simplicity, makes it a basis for designing database schemes conforming to many types of data models including relational and network models. In this section, we show via examples how SchemaLog, properly extended, can be used to query as well as restructure ER databases. In Appendix C, we provide a formal account of the

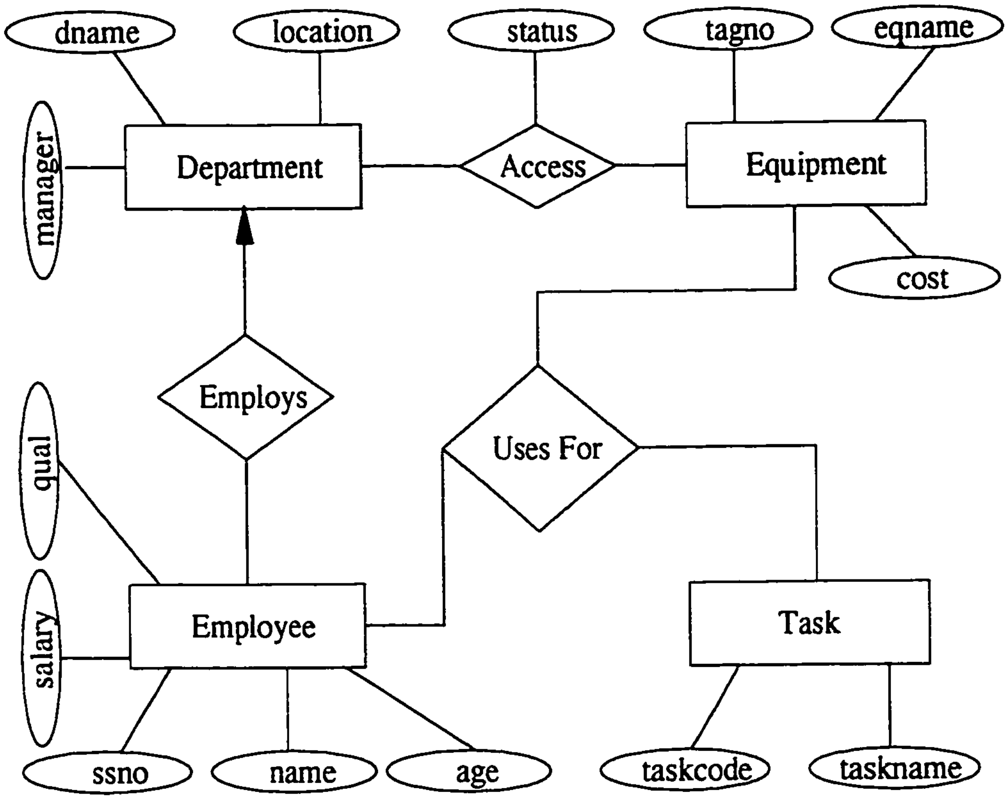


Figure 13: ER Diagram

syntax and semantics of the extended SchemaLog language.

The starting point of our approach to deploying SchemaLog for (extended) ER databases is the work of Grant, Ling, and Lee, who propose a logic for databases called *ERL* [GLL93]. ERL is to ER databases, what first-order logic is to relational databases and exhibits the following two important features: *(a)* ERL allows for predicates as arguments of predicates and *(b)* permits attributes that may be single-valued, multi-valued, or composite. Given an ER diagram of an application, [GLL93] provides a methodology for representing it in ERL. We illustrate the idea using the following example adapted from [GLL93].

Example 5.2.1 Consider the ER-diagram of Figure 13.

The ERL schema of this ER-diagram is the following:

department (dname, location, manager).
employee (ssno, name, age, salary, qual).
equipment(tagno, eqname, cost).

task(taskcode, taskname).
employs(department, employee).
access (status, department, equipment).
usesfor (employee, equipment, task).

Note that the predicates *department, equipment, task,* and *employee* also appear as arguments of other predicates.

5.2.1 Adapting SchemaLog to the (Extended) ER Model

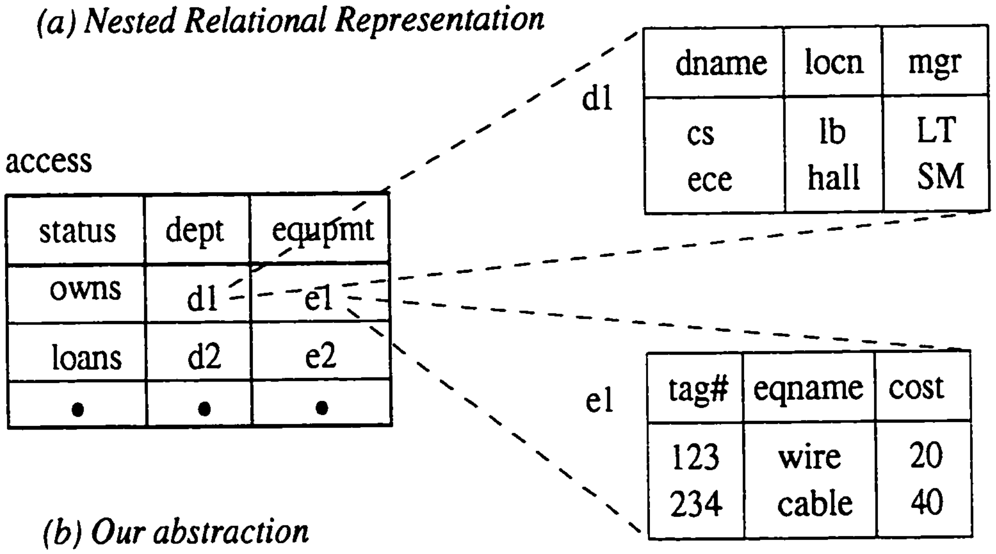
As illustrated in the above example, the idea behind mapping an ER diagram to a ERL schema is based on the following translation methodology. Corresponding to each entity and each regular and weak relationship in the ER diagram, there is a predicate symbol in the ERL schema. Moreover, the set of attributes of the predicate corresponding to a relationship, contains the names of the entities involved in the relationship. Notice that such a relation can be thought of as a nested relation – the 'inner relations' being the relations corresponding to the entities (that stand in the relationship). For example, the nested relation corresponding to *access* would have a simple attribute *status* and composite attributes (which is a nested relation) corresponding to (entities) *dept* and *equipment*. Figure 14(a) illustrates this nested relation. Based on this view, we build our proposal for applying SchemaLog to ER databases.

SchemaLog, as presented in chapter 2 does not have the provision for modeling non-1NF relations. However, we note that by a slight modification (extension) to the syntax presented in Section 2.3.1, it is possible to adapt SchemaLog to handle complex values. The idea behind the extension is the following. We treat each complex value (a sub-relation) in a single tuple of a nested relation as a distinct relation (that exists independent of the parent relation, and hence can be thought of as a regular relation). Such an interpretation essentially corresponds to *(a)* identifying each sub-relation in a nested relation with a specific (atomic) identifier, and *(b)* using this identifier as the name of the relation to refer to it as though it existed as a regular relation. Figure 14(b) illustrates this idea of denoting the sub-relation with id's and reasoning about the nested relation using these id's. Some examples of queries are now in order.

Example 5.2.2 Consider the ER database of Example 5.2.1. We now give some access

status	department			equipment		
owns	dname	locn	mgr	tag#	eqname	cost
	cs ece	lb hall	LT SM	123 234	wire cable	20 40
loans	dname	locn	mgr	tag#	eqname	cost
•	•	•	•	•	•	•

(a) Nested Relational Representation



status	dept	equpmt
owns	d1	e1
loans	d2	e2
•	•	•

dname	locn	mgr
cs ece	lb hall	LT SM

tag#	eqname	cost
123 234	wire cable	20 40

(b) Our abstraction

Figure 14: The Access Relation

examples of queries in SchemaLog that can be posed against this database.

Q_1 : *Find the portion of the* access *relationship for the* status *value* owns

This can be expressed in the (extended) SchemaLog as:

$\longleftarrow erdb :: access[status \rightarrow `owns', \ department \rightarrow D, \ equipment \rightarrow E],$
$\quad D[dname \rightarrow NAME, \ location \rightarrow LOCN, \ manager \rightarrow MGR],$
$\quad E[tagno \rightarrow TAG\#, \ eqname \rightarrow EQNM, \ cost \rightarrow COST].$

In this SchemaLog query, variables D and E range over (atomic) id's corresponding to the subrelations in the *access* relation. D ranges over the id's corresponding to each distinct department that has status 'owns'. We interpret these id's as the names of the relations that contain (the compound) values pertaining to the departments, and access them as though they existed as independent relations. Notice how a new type of atom (derived from the SchemaLog atom of depth 4, by stripping off the database name place-holder) is used to refer to these nested relations. The new atom is required to distinguish the nested relations from that of the normal database relations.

Q_2 : *Find the* tag# *and* cost *of all equipment used for the task with code* XYZ

This query can be posed as:

$\longleftarrow erdb :: usesfor[task \rightarrow T, \ equipment \rightarrow E],$
$\quad E[tagno \rightarrow TAG\#, \ cost \rightarrow COST], \ T[taskcode \rightarrow `\text{XYZ}'].$

As seen in the above examples, SchemaLog with a minor modification involving the introduction of a new 'db-less atom', is capable of querying ER databases. Despite this change in the syntax, the semantics of the language requires little modification. In Appendix C, we provide a formal account of the syntax and semantics of the extended SchemaLog language.

We remark that the the ability of SchemaLog to cater to the (extended) ER model, one of the most fundamental models of database systems, is a testimony to the generality and usefulness of a SchemaLog-based approach to querying and restructuring information. A case in point is the nested-relational model which, as the reader can readily see, can easily be modeled in SchemaLog using the approach described in this section.

5.3 SchemaLog in the Context of Querying and Restructuring the World Wide Web

The World Wide Web (WWW) is revolutionizing the information age. Its strong impact on the end user and the many potential benefits it augurs have spurred tremendous research on a whole gamut of issues related to storing, retrieving, and manipulating information on the Web. The Web is emerging to be one of the most exciting topics of active research, bringing together researchers from diverse areas such as communications, electronic publishing, language processing, and databases.

While the Web is a resource of colossal amount of information, the task of searching for specific information of one's interest is difficult and is fraught with the danger of the user being overwhelmed by a deluge of information. This is typically the scenario that a casual user encounters while using any of the available (keyword based) Web search engines. Besides, users often have partial knowledge on the information (such as the layout of the page(s), patterns appearing in it, or synonyms of words or sentences in the page(s)), they are looking for. However, currently available tools do not provide a facility for expressing such partial knowledge the users may have on the information they are querying. Yet another need experienced by an emerging class of users (commonly referred to as *information-brokers*) pertains to pooling together information from multiple sources and integrating it in a common source. Again. available tools do not provide adequate support for such consolidation of information.

Undeterred by these obvious shortcomings and propped up by some ad-hoc tools and primitive techniques. the Web is growing in an explosive way due to its usefulness even with these restrictions. However, unless tools and techniques based on sound formal foundations, for catering to the above needs are developed, the Web's full potential cannot be realized. In this context, the experience of successfully applying SchemaLog in the context of non-relational database systems providing the requisite verve, we investigate how the powerful features of the our language can be put to use in the context of querying and restructuring information on the Web.

Thus, the problem we address is the following: *How can a user (seeking specific information she is interested in)* retrieve *and perhaps* restructure *information in the Web?* In real life, she might also have clues on the information of interest to her, such as its likely location in the Web, its structure, some keywords, or patterns relevant to it. We would like to enable the user to specify such knowledge as part of the query, make use of it to search the Web, and return a helpful response "satisfying" the query.

Most information in the Web is present in the form of Hypertext Markup Language (HTML) documents. Our proposal in this section, provides for a *declarative* way for *retrieving* and *restructuring* information in HTML documents. In the subsequent discussions, we use the terms Web documents and HTML documents interchangeably.

Querying the Web – State of the Art: The compelling need for querying information in the Web has led to the development of a number of tools that, based on some keywords specified by the user, search the Web and return information related to the keywords. *Lycos*, *WWWW* (World Wide Web Worm), *NetFind*, and *InfoSeek* are some examples. These search tools make use of a pre-compiled *catalog* (also called a *reference database*) of information available in the Web to answer user queries. Searches can be performed on titles, reference hypertext, URL[1] etc. These tools typically have two components: *resource locator* and *search interface*. The resource locator is run periodically to gather information from the Web and create the catalog. The search interface provides fast access to information in the catalog. In Section 5.3.5, we discuss in detail two popular search tools – WWWW, and Lycos. More information on Web search tools in general can be found in [SEKN92, Kos]. Currently available Web search tools suffer from the following drawbacks.

- *Partial knowledge that a user might have on the information she is querying about is not fully exploited.*

  As an example, consider the user searching for call for papers (CFPs) of all database conferences. In particular, she is interested in knowing the submission deadlines of various conferences. She also has the following information on CFPs.
  (i) Most database related CFP's can be obtained by *navigating links* from the page of Michael Ley at http://www.informatik.uni-trier.de/~ley/db/index.html.
  (ii) Each CFP has a pattern of the form "... Date ... ⟨*date*⟩ .." (or a 'synonym' of this pattern), where ⟨*date*⟩ is potentially the submission deadline.

  We would like the user to be able to express such information when she specifies

[1]Uniform Resource Locator, an address that specifies the location of a resource in the Web.

the query. The query processor should also make use of this information to perform an efficient search.

- *The restructuring ability these tools provide is limited, or non-existent.*

  Continuing on the example above, the user might want to group together links to CFPs that have submission deadlines in the same month. *The querying tool should allow for specifying in an ad hoc manner the format in which the answer should be presented.*

- *There are no facilities for exploiting externally available libraries of string and document processing functions.*

  Various resources for processing documents and strings are available in the form of external functions. The querying medium should be able to exploit these readily available tools for allowing powerful and natural ways of querying the Web.

- *Query result quality is compromised by the highly dynamic nature of Web documents.*

  In most search tools, once the page is visited, it is marked read and never visited again (unless explicitly asked to). But, by its very nature, information in each document is ever changing/expanding along with the Web. Thus, soon after an update. the catalog could become out of date. The search engines thus do not take into account the highly dynamic nature of the Web.

Besides these shortcomings. the search interface provided by these tools is highly restrictive. A typical interface would ask for keywords in the document the user is interested in and a choice of the kind of search to be done (*e.g.* title search, URL search, keyword search etc). Thus, the search possibilities are circumscribed by the limited choices provided by the search interface, as opposed to the possibilities opened up by ad hoc querying.

Given the parallels between many of the needs on the Web and those traditionally addressed by the database community, many database researchers ([LSS96a, LRO96, MMM96, AMMT96, Abi97]) have realized that tools and techniques well-studied for databases can be brought to bear in the context of the Web. In particular, in the area of structured document querying – a broader paradigm of which Web documents can be thought of as a special case – there have been early ([SHT+77]) as well as recent ([ACM93, CAS94, ACM95]) works that base their approach on providing a "database view" of the structured information present in documents. Most proposals in this direction follow the idea of "mapping" the underlying grammar of the document to an appropriate database scheme. Thus, when the document is parsed using this grammar, corresponding objects would be created in the database. These works exploit the well studied optimization techniques in databases to improve "document query processing". In Section 5.3.5, we discuss some of these works in more detail.

Though the Web is a collection of structured documents, the nature of interrelationships among these documents raises new issues that are yet to be addressed in a convincing manner by the above proposals for document query processing. Information gathering in the Web lays its emphasis on *navigation* via hyperlinks that relate documents to one another. The above proposals do not account for the notion of hyperlinks and the associated aspects of navigation. Restructuring the relationship among the various documents is another issue yet to be addressed satisfactorily. Also, the multimedia nature of (certain) Web documents, calls for novel ideas and techniques to address the problem of querying the Web.

Our Strategy: Inspired by the SchemaLog experience, we adopt a pragmatic approach to the problem of querying and restructuring information in the Web. We propose a declarative query language called *WebLog* that has its roots in SchemaLog. Some of the highlights of *WebLog* are (a) providing a *declarative* interface for querying as well as restructuring (the language is logic based), (b) accommodating *partial knowledge a users may have* on the information being queried (*WebLog* has a rich syntax and semantics), (c) providing ways for *seamlessly integrating libraries of document analysis and string processing algorithms*, thus putting their combined power to work for querying and restructuring the Web (the language has facilities for specifying foreign functions in the form of 'built-in predicates'), and (d) recognizing the *dynamic nature of Web information* (query processing need not be done on catalogs, but on the Web itself by taking advantage of 'navigation landmarks' the user has the flexibility to specify).

5.3.1 HTML Overview

In this section, we present the main features of the Hypertext Markup Language (HTML), the language of most Web documents today. A detailed discussion is clearly beyond the scope of this thesis. Interested readers are referred to [BC95].

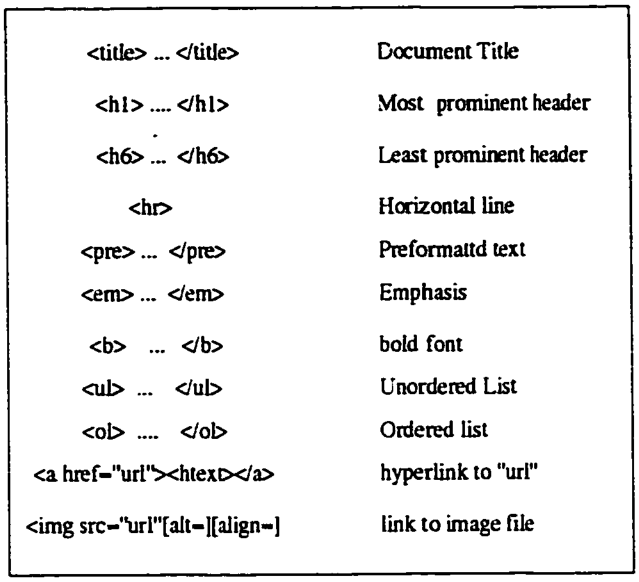
HTML is the language used for creating hypertext documents on the Web. It has the facility for creating documents that contain *hyperlinks* which are pointers from keywords appearing in the document to a destination. At its simplest, the destination is another HTML document. The destination could also be a resource such as an external image. a video clip, or a sound file. Hyperlinks are the most important constituent of HTML documents. They are composed of two *anchors* - a source anchor (henceforth called *hypertext*) that specifies the start of the hypertext link and a destination anchor (henceforth called *href*) that is a pointer to the document to be linked from the source. The display that is the result of viewing a HTML document using a browser (such as NCSA Mosaic or NetScape) is called a *page*. The hypertext appears as a highlighted text in the page. Activation (usually clicking) of hypertext is interpreted as a request for displaying the destination document. We call this process *navigation*. For reasons that will become obvious in Section 5.3.2, we associate an id called a *hlink-id* with each hyperlink. A hlink-id has two components, the hypertext, and the href.

HTML allows for preparing documents for Web browsing by embedding control codes (*tags*) in ASCII text to designate titles, headings, paragraphs. and hyperlinks. Figure 15 contains some of the commonly used tags in a HTML document.

Conceptually. an HTML document consists of two parts: the *head* and the *body*. The head contains meta-information about the document. It is specified using the tag '$\langle title \rangle$'. The body consists of the document contents that includes headings. text. images, voice, video etc and hyperlinks. Users can navigate over the various documents by activating the hyperlink that would be of interest. Figure 16 shows a sample HTML document at URL: http://www.informatik.uni-trier.de/~ley/db/index.html, and Figure 17 shows the corresponding page viewed using the browser NCSA Mosaic.

5.3.2 Conceptual Model

In this section, we describe the *WebLog* model of HTML information. The conceptual model suggests simple, yet powerful ways of querying HTML documents. It also



<title> ... </title>	Document Title
<h1> .... </h1>	Most prominent header
<h6> ... </h6>	Least prominent header
<hr>	Horizontal line
<pre> ... </pre>	Preformattd text
<em> ... </em>	Emphasis
<b> ... </b>	bold font
<ul> ... </ul>	Unordered List
<ol> .... </ol>	Ordered list
<a href="url"><htext></a>	hyperlink to "url"
<img src="url"[alt=][align=]	link to image file

Figure 15: Common Tags in HTML

facilitates making use of common knowledge users might have about a document.

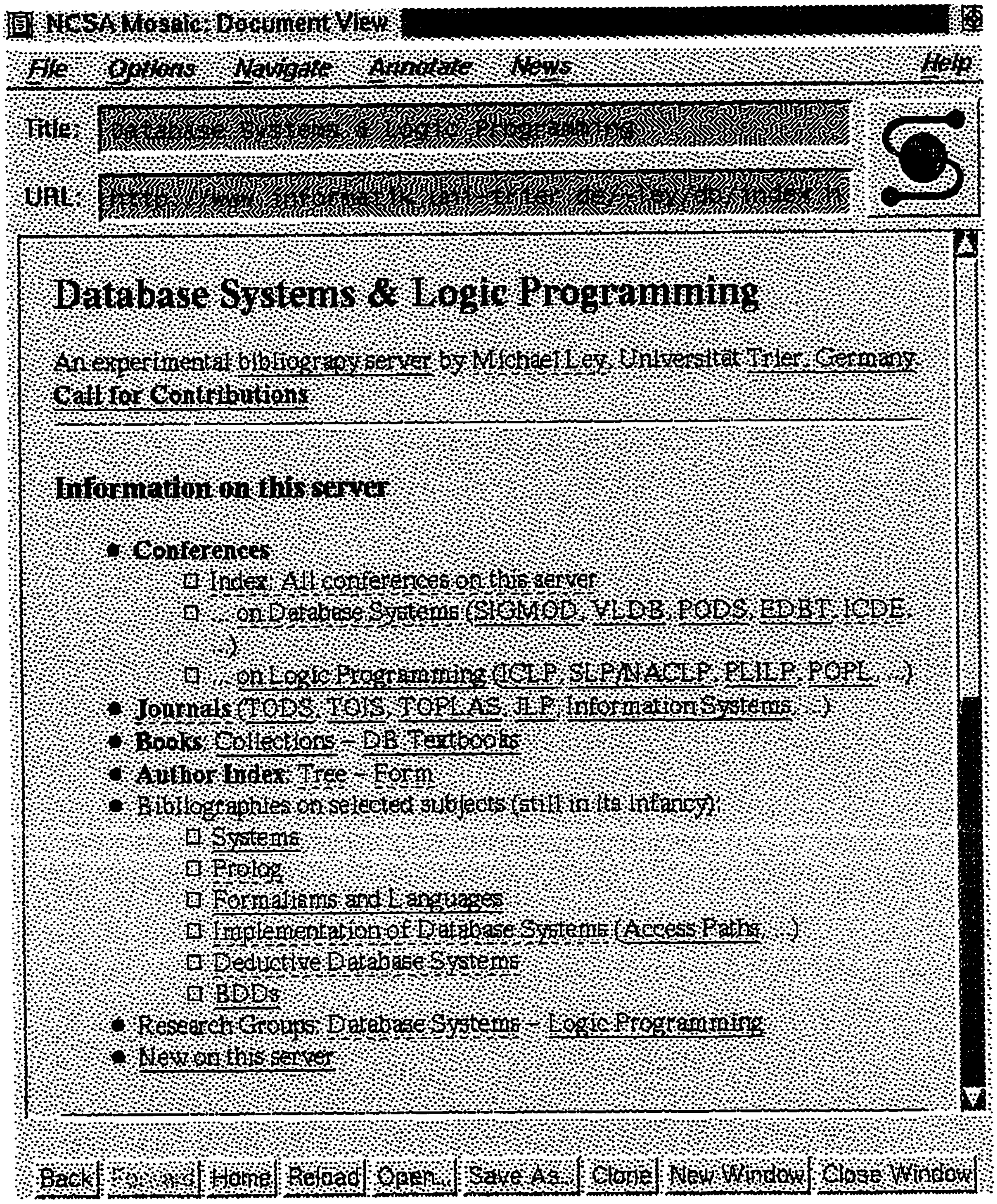
Each page (and hence a document) consists of a heterogeneous mix of information about the topic mentioned in the 'title' of the document. In practice, a typical document would consist of "groups of related information" that are spatially close together in the page. Information within each such group would be homogeneous. For instance, information enclosed within the tag <HR> (horizontal line) in a document could form a group of related information. Similarly, the tags header, paragraphs, lists etc could play the role of delimiting one group from another.

We would like to distinguish between groups of related information appearing in a page. We call each such group, a *rel-infon. A page is thus a set of rel-infons.*

The notion of what constitutes a rel-infon is highly subjective. We believe this choice should be left to the user, who will define it based on her needs. For example, in the HTML document presented in Figure 16, we could consider either the tag <HR> or <UL> to be the rel-infon delimiter. In the former case, the granularity we obtain for a rel-infon is at the level of distinguishing information present in the Ley server, and elsewhere. In the latter case, the granularity is finer – information appearing under 'conferences', 'journals' etc would be considered as corresponding to different rel-infons. We argue that this flexibility of specifying rel-infon granularity is a source of great power in expressing queries.

```
<html><head><title>Database Systems &amp; Logic Programming</title></head>
<body><h1>Database Systems &amp; Logic Programming</h1>
An experimental
<a href="intro.html">bibliograpy server</a> by
<a href="http://www.informatik.uni-trier.de/~ley/addr.html">Michael Ley</a>,
Universit&auml;t
<a href="http://www.uni-trier.de/trier/trier_eng.html">Trier, Germany</a>.<br>
<b><a href="about/call.html">Call for Contributions</a></b>

<hr>

<h2>Information on this server</h2>
<ul>
<li><b>Conferences</b>
<ul>
<li><a href="conf/index.a.html">Index: All conferences on this server</a>
<li>... <a href="conf/index.html">on Database Systems</a>
(<a href="conf/sigmod/index.html">SIGMOD</a>,
<a href="conf/vldb/index.html">VLDB</a>,
...)
<li>... <a href="conf/index.l.html">on Logic Programming</a>
(<a href="conf/iclp/index.html">ICLP</a>,
<a href="conf/slp/index.html">SLP/NACLP</a>,
...)
</ul>
<li><a href="journals/index.html"><b>Journals</b></a>
(<a href="journals/tods/index.html">TODS</a>,
<a href="journals/tois/index.html">TOIS</a>,
...)
</ul>

<hr>

<h2>Links to related services</h2>
<ul>
<li><a href="../organizations.html">Computer Science Organizations</a>
(<a href="http://info.acm.org/">ACM</a> -
<a href="http://bunny.cs.uiuc.edu/README.html">SIGMOD</a> -
<a href="http://info.sigir.acm.org/sigir/">SIGIR</a> -
<a href="http://www.cs.mu.oz.au/~ad/alp/info-alp.html">ALP</a> -
etc.)
<li><a href="http://www.comlab.ox.ac.uk/archive/logic-prog.html">WWW
Virtual Library: Logic Programming</a> (by Jonathan Bowen, Oxford)
<li><a href="http://web.cs.city.ac.uk/archive/constraints/constraints.html">City
University Constraints Archive</a>
</ul>

<hr>
<address>
<a href="http://www.informatik.uni-trier.de/~ley/addr.html">Michael Ley</a>
(ley@uni-trier.de)
19-Jul-95
</address>
</body>
</html>
```

Figure 16: Sample HTML Code


NCSA Mosaic: Document View
File Options Navigate Annotate News Help
Title: Database Systems & Logic Programming
URL: http://www.informatik.uni-trier.de/~ley/db/index.h
Back Forward Home Reload Open... Save As... Clone New Window Close Window

Database Systems & Logic Programming

An experimental bibliograpy server by Michael Ley, Universität Trier, Germany.
Call for Contributions

Information on this server

- **Conferences**
  - Index: All conferences on this server
  - ... on Database Systems (SIGMOD, VLDB, PODS, EDBT, ICDE, ...)
  - ... on Logic Programming (ICLP, SLP/NACLP, PLILP, POPL, ...)
- **Journals** (TODS, TOIS, TOPLAS, JLP, Information Systems, ...)
- **Books**: Collections – DB Textbooks
- **Author Index**: Tree – Form
- Bibliographies on selected subjects (still in its infancy):
  - Systems
  - Prolog
  - Formalisms and Languages
  - Implementation of Database Systems (Access Paths, ...)
  - Deductive Database Systems
  - BDDs
- Research Groups: Database Systems – Logic Programming
- New on this server

Figure 17: The page at leyurl

From the perspective of querying and restructuring HTML documents, the information that would be of utmost interest in a page are keywords or more generally strings, hyperlinks, and tags that adorn strings. We would like to provide 'first-class status' to all these concepts in our model.

A rel-infon has several *attributes.* The attributes come from a set consisting of strings 'occurs', 'hlink', and various tags (such as `<title>`, `<b>`, `<em>`) that adorn strings in a HTML document. The attributes of a rel-infon map to 'values' that are strings, except for the hlink attribute that is mapped to a hlink-id.

Formally, let $\mathcal{T}$ be the set of all tags that adorn tokens in a HTML document, $\mathcal{S}$ be the set of strings, and $\mathcal{H}$ be the set of hlink-ids. A rel-infon is a partial, set valued mapping I.

$$I \ : \ \{`occurs', `hlink'\} \cup \mathcal{T} \ \rightarrow \ 2^{\mathcal{S} \cup \mathcal{H}}$$

Intuitively, the attributes are meant to play the following role in modeling a rel-infon. The attribute 'occurs' is mapped to the set of strings occurring in a rel-infon: 'hlink' is mapped to the set of hlink-id's of hyperlinks appearing in a rel-infon, and the tag attributes, if defined, are mapped to the tokens they adorn in the rel-infon. For instance, in the document in Figure 16, if we consider `<ul>` to be the rel-infon delimiter, $b \rightarrow `conferences'$ is a legal 'attribute/value pair' in the rel-infon on conferences. The tag attribute 'title' is a special one; it is mapped to the same string (the title of the document) regardless of the rel-infon in the document.

Each rel-infon also has a unique *id.* This id could be the a token appearing in a header associated with a rel-infon, the most prominent 'keyword' in a rel-infon, or even the byte offset of the start of the rel-infon from the beginning of the page. Our model does not commit to a specific choice and offers some flexibility.

5.3.3 *WebLog* – Syntax and Semantics

In this section, we discuss the syntax and semantics of *WebLog.* We also discuss the role of built-in predicates in a *WebLog* programming environment.

Syntax

As mentioned earlier, *WebLog* is inspired by SchemaLog and its syntax can be seen to be a 'clone' of that of SchemaLog. We use strings starting with a lower case letter for constants and those starting with an upper case letter for variables. As a special case, we use t_i to denote arbitrary terms of the language. $\mathcal{A}, \mathcal{B}$, ... denote arbitrary well-formed formulas and A, B, ... denote arbitrary atoms.

The vocabulary of *WebLog* consists of pairwise disjoint countable sets $\mathcal{G}$ (of function symbols), $\mathcal{S}$ (of non-function symbols), $\mathcal{V}$ (of variables), and the usual logical connectives $\neg, \vee, \wedge, \exists$, and $\forall$.

Every symbol in $\mathcal{S} \cup \mathcal{V}$ is a term of the language. If $f \in \mathcal{G}$ is a n-place function symbol, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

An *atomic formula* of *WebLog* is an expression of one of the following forms:
$\langle\text{url}\rangle[\langle\text{rid}\rangle : \langle\text{attr}\rangle \twoheadrightarrow \langle\text{val}\rangle]$
$\langle\text{url}\rangle[\langle\text{rid}\rangle : \langle\text{attr}\rangle \rightarrow \langle\text{val}\rangle]$
$\langle\text{url}\rangle[\langle\text{attr}\rangle]$
$\langle\text{url}\rangle$
where $\langle\text{url}\rangle$, $\langle\text{attr}\rangle$, $\langle\text{rid}\rangle$, and $\langle\text{val}\rangle$ are terms of *WebLog*. We refer to them as *url term*, *attr term*, *rid term*, and *val term* respectively. The rid term intuitively stands for the rel-infon id (rid) and is optional. The well-formed formulas (wff's) of *WebLog* are defined as usual: every atom is a wff; $\neg\mathcal{A}, \mathcal{A} \vee \mathcal{B}, \mathcal{A} \wedge \mathcal{B}, (\exists X)\mathcal{A}$, and $(\forall X)\mathcal{A}$ are wff's of $\mathcal{L}$ whenever $\mathcal{A}$ and $\mathcal{B}$ are wff's and X is a variable.

We also permit *molecular formulas* of the form
$\langle\text{url}\rangle[\langle\text{rid}\rangle : \langle\text{attr}_1\rangle \twoheadrightarrow \langle\text{val}_1\rangle, \ldots, \langle\text{attr}_n\rangle \twoheadrightarrow \langle\text{val}_n\rangle]$ as an abbreviation of the corresponding well-formed formula
$\langle\text{url}\rangle[\langle\text{rid}\rangle : \langle\text{attr}_1\rangle \twoheadrightarrow \langle\text{val}_1\rangle] \wedge \cdots \wedge \langle\text{url}\rangle[\langle\text{rid}\rangle : \langle\text{attr}_n\rangle \twoheadrightarrow \langle\text{val}_n\rangle]$. In spirit, this is similar to the molecules in SchemaLog.

Example: $http : //www.com[X : title \rightarrow \text{'}Web\text{'}, hlink \twoheadrightarrow L, occurs \twoheadrightarrow \text{'}example\text{'}]$ is an atomic formula in *WebLog*. Here, the url address is the url term, *title*, *hlink*, and *occurs* are the attr terms. X is the rid term, and the remaining terms are the val terms.

Next, we present the semantics of *WebLog* using examples. The examples make use of "built-in predicates" that are tailor-made for the Web setting. Built-ins play a significant role in a *WebLog* programming environment. Before we present the semantics, we discuss some of the commonly used built-in predicates.

Built-in Predicates

In any application, it is often useful (or necessary) to express certain general relationships whose semantics is well understood in the context of the application. Thus, these relationships need not be explicitly defined in the program, but are implicitly known to the system. Such relationships are expressed using special predicates called *built-in predicates.* Some examples of built-ins in datalog are the arithmetic predicates ($<, >, =$ etc).

In the context of querying the Web, there is a natural need for string processing and for conventional and multimedia document analysis. In real life, algorithms for such analysis are available as stand-alone, external functions. With an aim towards exploiting the availability of these external resources, *we treat built-in predicates as (abstractions of) external functions.* This novel approach (a) combines the power of search, deduction, and external functions, and (b) makes it easy to (continually) incorporate new external functions even as they are deployed. Built-ins are a source of great power for *WebLog.* Naturally, our treatment of built-in predicates calls for the notion of a 'legal binding pattern' – the external function's pre-condition with regard to an argument being either *bound* or *free* – with which a built-in could be invoked.

In the following, we identify some useful built-in predicates for *WebLog* applications. All of these built-ins could be invoked with a binding pattern of either (*bound, free*) or (*bound, bound*).

- *href(< hlink-id >, < url >):* This predicate captures the relationship between a hlink-id and the destination anchor in its corresponding hyperlink. The second argument stands for the URL of the destination.

- *htext(< hlink-id >, < string >):* This is the counterpart of *href* that captures the relationship between a hyperlink and its source anchor. The second argument stands for the *hypertext* that is the source anchor in the hyperlink corresponding to the < *hlink-id* >.

- *substring(< string >, < string >):* Pattern matching is an important requirement in document querying. The binary built-in predicate *substring*, is useful in this context. The first argument of *substring* is the source string and the second argument is a substring in the source.

- *isa(< string >, < type >):* This predicate is useful for type checking – a need often felt while querying patterns in documents. The first argument of isa is some object that is represented as a string. The second argument is its type (*e.g.* int, float, string, date, url, year etc.). While we do not anticipate a need for sophisticated type checking as in OO databases or in OO programming, the algorithms implementing this foreign function could use a predetermined library to assign types to values. For example, the string following the keyword date: is assigned the type date.

- *len(< string >, int):* The second argument of this predicate is the length of the string in its first argument.

- *newlink(< string >, < hlink-id >):* The second argument is a unique hlink-id corresponding to the string occurring in the first argument. The htext component of this hlink-id is the string itself, while the href component is system generated. This predicate is useful for generating new hlink-ids in *WebLog.*

Besides these commonly used built-ins, we would freely make use of other useful predicates (such as *synonym*, *homonym* etc.) whose semantics would be clear from the context. In particular, built-ins that facilitate querying multimedia documents as abstractions of appropriate media processing algorithms would be extremely useful.

Programming Predicates: In the context of queries as well as view definitions, as in SchemaLog, it will be convenient to have the facility for programming predicates *i.e.* predicates which do not refer to any document, but exist only in the context of a program. We shall freely make use of programming predicates in the rest of the examples in this section.

Semantics

In this section, we informally present the semantics of *WebLog.* We make use of real life examples in our presentation. Through these examples we will illustrate the power of *WebLog* to *(a)* navigate hyperlinks, *(b)* search titles as well as the document for keywords, *(c)* recognize patterns appearing in documents, and *(d)* perform restructuring. We use the Ley server originating at the "Database Systems & Logic Programming" page (`http://www.informatik.uni-trier.de/~ley/db/index.html`[2]) for our illustration.

The semantics closely follows the conceptual model. The *url term* stands for the URL of a page, and similarly the *attr term*, *rid term* and *val term* stand for the concept they are named after. Symbols '[' and ']' in the syntax, enclose attribute/value pairs in the context of a single rel-infon in the document. An assertion $url[rid : attr \twoheadrightarrow value]$ is interpreted as saying *value* belongs to the set of values associated with *attr* in the context of the rel-infon *rid* in the page *url*. We will illustrate these notions via examples.

Hyperlink Navigation and Searching Titles

One of the novel features of *WebLog* is that it treats hyperlinks as 'first class citizens'. This provides the facility for navigating across HTML documents using a *WebLog* program. Hyperlinks can also be queried like ordinary data, and used for restructuring. The following example illustrates these ideas.

(Q_1) *We are interested in collecting all citations (hyperlinks) referring to HTML documents, that appear in the Database Systems & Logic Programming page. We would also like this collection to contain the title of the document the citation refers to.* The following *WebLog* program expresses this need.

$$
\begin{aligned}
&ans.html[title \rightarrow \text{`all citations'}, hlink \twoheadrightarrow L, occurs \twoheadrightarrow T] \\
&\quad \longleftarrow leyurl[hlink \twoheadrightarrow L],\ href(L, U),\ U[title \rightarrow T].
\end{aligned}
$$

Variable L in the first subgoal ranges over all hyperlinks in *leyurl*. The built-in predicate *href* is used to navigate over the citations in the page at *leyurl*. The rule generates a new HTML document *ans.html* that is a collection of all citations in *leyurl*, annotated with the title of the cited document.

The navigation in this example is simple; there is just one level of traversal. Navigation of a more general kind is illustrated in the next example.

[2]In order to avoid repeating this long URL, from now on we will refer to it as *leyurl*.

Querying Keywords in Documents

The Ley server is the collection of documents originating in the *leyurl*. These documents have the property that *(i)* they can be reached by navigating links originating in the *leyurl*, and *(ii)* their URL will have the prefix `http://www.informatik.uni-trier.de/ĩey/`. Suppose we would like to make use of this knowledge to express the following query:

(Q_2) *Find all documents in the Ley server that have information related to 'Interoperability'.*

The *WebLog* program that expresses this query is:

$ley\_server\_pages(http://www.informatik.uni\text{-}trier.de/\tilde{}ley/db/index.html) \longleftarrow$

$ley\_server\_pages(U) \longleftarrow ley\_server\_pages(V), V[hlink \twoheadrightarrow L],\ href(L, U),$

$\quad substring(U, http://www.informatik.uni\text{-}trier.de/\tilde{}ley/).$

$interesting\_urls(U) \longleftarrow ley\_server\_pages(U),$

$\quad U[occurs \twoheadrightarrow I],\ synonym(I, \text{'}Interoperability\text{'}).$

Rules (1) and (2) help identify the documents belonging to *Ley server*. The recursive rule (Rule (2)) essentially captures the properties *(i), and (ii)*, known to the user. Navigation is done via recursion. Rule (3) searches for occurrences of keywords related to 'Interoperability' (captured using the predicate *synonym*) in the Ley server documents and returns the relevant URLs in the relation *interesting_url*. The predicates *ley_server_pages(U)* and *interesting_urls(U)* are programming predicates; *href*, *substring*, and *synonym* are built-ins.

Querying Patterns

The following example illustrates how patterns appearing in HTML documents can be conventionally queried in *WebLog*. It also demonstrates the handling of types in our framework.

(Q_3) *Suppose we know that a paper on Coral*[3] *has appeared in the* VLDB Journal. *We do not know which year this paper appeared, but would like to find this information.*

[3]A deductive database system from University of Wisconsin, Madison

We know that a bibliography of papers on Coral can be found in a document with title 'Coral', accessible from the Ley server. We of course, know that the year would appear in the bibliography entry. The following rule in *WebLog* helps find out the year.

$$ans(Y) \longleftarrow ley\_server\_pages(U), U[title \rightarrow \text{'}Coral\text{'}, occurs \twoheadrightarrow S], len(S, 40),$$
$$substring(S, \text{'}VLDB\ Journal\text{'}), substring(S, Y),\ isa(Y, year).$$

---

ley_server_pages is the relation containing the URL's of documents in Ley server (obtained using the program for query Q_2). The body of the above rule expresses the user's knowledge that *(a)* the title of the bibliography document is 'Coral', *(b)* It has some string (whose length we specify is 40) that has substrings '*VLDB Journal*' and the string that stands for the year. We could also express our knowledge that the string corresponding to the year is of type *year*. Thus, the answer relation would contain all years that appear in a string of length 40 that has *VLDB Journal* as the substring. One of these integers must be the year in which this paper appeared.

Restructuring

A simple instance of restructuring using *WebLog* can be seen in the program for query Q_1 in Section 5.3.3, where the answer is presented as an HTML document containing the appropriate citations. In this section, we illustrate via examples, the sophisticated restructuring capabilities of *WebLog*.

(Q_4) *We would like to compile the citations of CFP's of all conferences having 'interoperability' as a topic of interest. We would also like to include information on the submission deadline in this compilation.*

We are aware that the CFP's can be obtained by navigating the tree of pages that has a root in the rel-infon containing the string 'conference', in *leyurl*. We also know that CFP's have patterns of the form '..submission...< *date* >..'. With this knowledge, Q_4 can be expressed the following way.

$$traverse(L) \longleftarrow leyurl[occurs \twoheadrightarrow \text{'}Conference\text{'}, hlink \twoheadrightarrow L].$$
$$traverse(L) \longleftarrow traverse(M),\ href(M, U),$$
$$U[occurs \twoheadrightarrow \text{'}Conference\text{'}],\ U[hlink \twoheadrightarrow L].$$

$cfp.html[L : title \rightarrow \text{'allcfps'},\ hlink \twoheadrightarrow L, occurs \twoheadrightarrow \text{'submission date :'}.D] \longleftarrow$
$\quad traverse(L),\ href(L,U),\ U[occurs \twoheadrightarrow \text{'Interoperability'}],\ U[occurs \twoheadrightarrow P],$
$\quad len(P,20),\ substring(P,S), synonym(S,\text{'submit'}),\ substring(P,D), isa(D,date).$

---

traverse(L) asserts the fact that the page cited via hyperlink L is traversed (from a page 'descending' from the *leyurl* page). Rule (1) initiates the navigation from the *leyurl* page via all hyperlinks that are present in the rel-infon having the keyword 'Conference'. Rule (2) is recursive, and is guaranteed to terminate soon because of the presence of the keyword 'Conference' in the second subgoal. Thus, only pages having this keyword would be traversed. The last rule uses the following idea to generate the 'result page'. If a page has the keyword 'Interoperability' and some rel-infon in the same page has a pattern that mentions 'submit' (or some synonym of that) along with a date in it, we infer it must be a CFP page that is of interest to us. Note the use of rid term L in the head. It helps in 'grouping' together the date and hlink information belonging to the same rel-infon in the restructured output. Note also the use of function symbol '.' (concatenation) in the head. It helps in appropriately positioning the strings in the output.

(Q_5) *Suppose we would like to restructure the newly generated cfp.html further in such a way that all conferences having a deadline in the same week are grouped together in a page.*

The following two rules help obtain the desired effect. (We assume a built-in predicate *dates2weeks* that converts a date to its corresponding week in the year.)

$U[title \twoheadrightarrow W, hlink \twoheadrightarrow L,\ occurs \twoheadrightarrow \text{'date :'}.D] \longleftarrow$
$\quad cfp.html[hlink \twoheadrightarrow L,\ occurs \twoheadrightarrow S],\ substring(S,D),\ isa(D,date),$
$\quad dates2weeks(D,W),\ newlink(W,M),\ href(M,U).$

$cfp\_by\_week.html[W{:}title \rightarrow \text{'byweek'},\ occurs \twoheadrightarrow \text{'weekNo.'},\ hlink \twoheadrightarrow M] \longleftarrow$
$\quad cfp.html[occurs \twoheadrightarrow S],\ substring(S,D),\ isa(D,date),\ dates2weeks(D,W),$
$\quad newlink(W,M).$

The first rule generates as many HTML documents as are distinct number of week numbers corresponding to the dates in cfp.html. Thus, links to CFP's having deadline in the same week are put together in the same page. For each week number, subgoal newlink generates a unique hyperlink, in whose location the CFP citations are added.

The second rule is used to generate a page that is an 'interface', containing links to the new set of pages that are generated in the first rule. In this rule, the rid term W is used to group together (as one rel-infon), the hlinks of the cfp pages having the deadline in the same week.

5.3.4 Discussions on Safety

In Chapter 3, we investigated the various notions of safety associated with a calculus language inspired by the querying fragment of SchemaLog and established equivalence results between each safe fragment of the calculus and an algebra capable of querying data as well as schema. In the context of the Web, the notion of safety assumes practical significance, worthy of further study. For instance, consider the query '$\longleftarrow X$' that requests a listing of all the URLs in the Web. Though this query is considered safe in the sense of the conventional notion of safety, for practical reasons of 'incomputability on the web' it should be considered unsafe.

We remark that the discussions as well as the results on safety discussed in Chapter 3, naturally extend to *WebLog* as well. Indeed, these notions have an important impact on the realization of a *WebLog*-based system for Web querying and restructuring. We do not discuss these issues further in this thesis, and plan to investigate them as part of future work.

5.3.5 Related Work and Discussion

This section discusses some important tools and techniques that have been proposed (and some of them available) for Web querying.

WWWW is a search tool developed at the University of Colorado by Oliver McBryan ([McB94]). WWWW has a resource locator that scours the Web inspecting all resources. Each HTML file found is indexed with its title string. Each URL referenced in a HTML file is indexed by the clickable hypertext associated with the URL, the name of the HTML file referring the URL, and its title. The information that is gathered by the locator is stored in four types of search databases - (1) citation hypertext, (2) citation URL, (3) HTML titles, and (4) HTML address databases. The search interface makes use of the Unix *egrep* program to query the catalog, and provides the option for searching each of these databases.

Lycos ([ML94]), developed at the Center for Machine Translation, Carnegie Mellon University is one of the popular Web search tool ([Poi95]). The Lycos resource locator (called, *web explorer*) searches the Web every day, building a database of all the Web pages it finds. The index of the catalog is updated every week. The explorer, written in Perl and C, provides the following information on each document to the catalog – title, headings and sub headings, 100 most weighty words, first 20 lines, and number of words. The search engine takes a user query, performs a retrieval from the catalog, and returns a list sorted according to a "match score". The engine is a C program that uses a disk-based inverted file retrieval system and a simple sum of weights to score documents.

As discussed in Section 5.3. these tools suffer from the drawbacks that *(a)* the ad hoc querying allowed against their rigid interface is limited, *(b)* there are no provisions for restructuring, *(c)* querying is done on a catalog that is difficult to keep up-to-date, and *(d)* there is little provision for exploiting users' partial knowledge.

Querying structured documents is currently an area of active research. Work in this area is relevant to the Web setting, given that that the Web documents are a special kind of structured documents. We now discuss some important works that apply database techniques in the context of querying documents.

The problem of extracting data from structured documents (including flat files) has been addressed by database researchers since the early days of the field ([SHT+77]). Recent advances in information modeling (*e.g.* semantically richer models such as the object-oriented model), and query processing (*e.g.* relational query optimization techniques) together with the need created by developments such as the Web, have stimulated fresh investigation of issues related to querying and updating structured data stored in documents ([ACM93. CAS94, ACM95]). In [ACM93], Abiteboul. Cluet, and Milo make use of the grammar of a document to 'map' it to an appropriate object-oriented database. They introduce the notion of a *structuring schema* which consists of the database schema and the grammar annotated with database programs that specify how terminals and non-terminals in the grammar are mapped to the schema. When the document is parsed, for each grammar rule that is fired, an appropriate instance, dictated by the annotation, is created in the database. They adopt well-studied database optimization techniques to efficiently perform this translation. Christophides, Abiteboul, Cluet, and Scholl ([CAS94]) use a similar idea to map SGML (Standard General Markup Language) documents into object-oriented databases. They also study the extensions to the object query languages (calculus, and SQL-like languages) necessitated by the mapping. Interestingly, in their model (as in ours') navigation is given a first class status using 'path variables'. More recently, Abiteboul, Cluet, and Milo ([ACM95]) study the (inverse) problem of propagating updates specified logically on a database, to a file that actually stores the semi-structured data. They investigate optimization techniques suitable for this "reverse translation".

Shortcomings of these proposals from a Web querying standpoint were identified in Section 5.3. On the other hand, we anticipate a natural use for techniques in these works in *WebLog* applications. We can use the techniques of [ACM93] to "materialize" relevant information into databases - OO or otherwise - which can then be queried using *WebLog*. We plan to investigate this issue in future.

Recent times have seen a spate of work, that make use of traditional database techniques to address the problem of querying the Web. W3QL ([KS95]) is a SQL-like query language designed specifically for the Web. This language has features for querying HTML documents and hyperlinks, but lacks restructuring facilities. HyperFile ([CGB95]) is an interesting data and query model useful in the context of querying the Web. The MultiSurf project ([MGE$^+$95]) is an attempt at integrating text browsing of a local database with hypertext browsing of the Web. [HVM95] presents the Hy+ system for graphical presentation and visual querying of structured data. Web visualization is the emphasis of Hy+. More recently, Mendelzon *et al.* [MMM96] propose a query language called WebSQL, that integrates textual retrieval with structure and topology based queries. They also propose a notion of 'locality' of queries - which addresses the question of *what fragment of the network* must be visited to answer a particular query. In [LRO96], Levy, Rajaraman, and Ordille describe their 'Information Manifold' system that provides a declarative description of the contents and capabilities of various information sources on the Web. They present algorithms that, based on the query as well as the descriptions of the sources, generate efficient query plans for processing the query. Closer to our approach is the work of Atzeni et. al. [AMMT96], on the 'Araneus data model' for Web documents that provides a relational abstraction on top of Web pages. Thus, in their framework, any relational query language can be used to query the Web. While our approach can be thought of as providing a syntactic relational abstraction (based on HTML) on top of web documents, [AMMT96] strives at providing a semantic abstraction that is document specific. However, they do not provide a methodology for specifying and realizing such relational views. Note that realizing such views in our context is straight-forward. Indeed, in our setting, semantic abstractions that are document specific can be built on top of the *WebLog* view, using a language such as SchemaLog.

Salient features of our framework include the following. The declarative query interface of *WebLog* facilitates simple and natural ways of querying Web information. Its rich syntax and semantics allows for expressing powerful queries that incorporate knowledge that users might have on the information being queried. *WebLog* provides first class status to hyperlinks. This novelty contributes to two major advantages – (*a*) the user can specify partial information on the traversals to be done to answer a query, and (*b*) the query processor can exploit this information to query directly on the Web in an efficient way (rather than query the catalog which might be out of date). This framework is also general enough to support multimedia information, as long as appropriate algorithms for media information processing and extracting their 'meta-data' are available for abstraction as built-in predicates.

We note that our proposal is *not* meant to replace the existing search facilities in the Web. We could build on the existing tools to realize our framework. In this sense, the work presented in this chapter complements the tools that are available for Web searching. For 'global' searches where the user has little knowledge about the information she wants (in terms of possible locations etc.), and wants to query the whole Web blindly, we still advocate the use of search engines. On the other hand, there are numerous circumstances where the user has partial knowledge of the information required. A compelling example of such a scenario arises in the context of the *Intranet*, where the structure of each page of the same 'kind' can be expected to have a certain degree of uniformity. In such cases, we anticipate the use of *WebLog* as an attractive alternative to search engines. Restructuring and ad hoc querying are the other unique advantages of *WebLog*.

5.4 Concluding Remarks

Legacy as well as non-traditional database systems constitute an important segment of information systems. In Chapter 2, we established SchemaLog as a powerful language that provides a fresh perspective on the notion of querying and restructuring relational databases and in the process provides for interoperability in a relational setting. In this chapter, we discussed how SchemaLog can be deployed in the context of non-relational information systems. In particular, we investigated how the novel features of SchemaLog such as *(a)* the ability to refer to schema information apart from data, and *(b)* the first-class treatment of all concepts, including the tid's, help us naturally interpret SchemaLog against Network databases, ER databases, and information repositories on the World Wide Web.

In real life, there is a need for interoperating among multiple information systems that belong to diverse data models. In this context, an ensemble of the various schemes we proposed in this chapter for naturally interpreting SchemaLog against different data models, would provide for an environment in which a *single* SchemaLog program would orchestrate interoperability among multiple information sources. For instance, we can write an application in SchemaLog that manipulates information from diverse sources such as relational databases, network databases, and the Web. This observation reinforces the importance of SchemaLog as a powerful language for developing advanced mediators that run on top of information sources, in a data model independent manner.

Chapter 6

Summary and Future Work

In this chapter, we recapitulate the main contributions of this thesis. We also identify and discuss the many exciting avenues for future research opened up by this thesis.

6.1 Summary

The objective of this thesis has been to study the foundations of the integration issues that arise in a federation of heterogeneous data sources. With this in mind, we developed an elegant logic called SchemaLog. We initially developed SchemaLog in the context of multiple relational databases and later extended it to cover non-relational sources (such as ER databases, Network databases, and information sources on the Web). SchemaLog treats the data in a data source as well as its schema components as first class citizens. This makes SchemaLog (syntactically) higher-order. We developed a simple first-order semantics for SchemaLog, based on the idea of making the intensions of higher-order objects explicit in the semantic structure and making the higher-order variables range over these intensions rather than the extensions they stand for. We also developed a fixpoint theoretic and proof-theoretic semantics of SchemaLog. In fact, the framework can be extended to incorporate the various forms of negation extensively studied in the literature of deductive databases and logic programming (see [She88] for a survey), notably stratified negation, without much difficulty. Even though SchemaLog is quite simple, our study (and our experience) indicates that it has a rich expressive power making it applicable to a variety of problems including interoperability, database programming (with schema browsing), schema integration and evolution, cooperative query answering, and powerful forms of aggregate computations, in the spirit of OLAP applications.

Realizing an efficient implementation of a SchemaLog-based system warrants the study of the calculus and algebraic languages underlying SchemaLog. We developed a new algebra by extending the conventional relational algebra with some new operations that are capable of manipulating both data and schema information in a federation of databases. In particular, the algebra has schema querying as well as schema restructuring operations. We also developed a calculus language inspired by SchemaLog. Based on the calculus language, we studied varying notions of safety applicable in a federation scenario, from the most liberal – in which queries could refer to any object in a federation, to the most restrictive – in which safe queries could refer to only a restricted subset of objects in a federation. We also studied in depth the relative expressive power of the various fragments of the algebra, the calculus, and a querying fragment of SchemaLog.

Practical relevance and impact on the industry have been some of the primary concerns of our research on the foundations of data integration. In this vein, in order to cater to the practitioners in the industry, we developed a principled extension of SQL in the spirit of SchemaLog, called *SchemaSQL* . *SchemaSQL* is downward compatible with SQL in that every SQL statement is a *SchemaSQL* statement having precisely the same semantics as the SQL statement. Besides, *SchemaSQL* treats data and meta-data in a uniform manner (*a la* SchemaLog) and allows the programmer to naturally express queries that manipulate both data and schema components of database systems. In particular, *SchemaSQL* is capable of *(a)* representing data in a database, in a structure substantially different from the original database, in which data and meta-data may be interchanged, *(b)* creating views whose schema is dynamically dependent on the input database, *(c)* expressing novel aggregation (over rows, and in general blocks of information) operations, in the spirit of some of the functionalities needed in OLAP applications, and *(d)* providing a great facility for interoperability and data, meta-data management in multidatabase systems. We described an architecture for realizing a *non-intrusive SchemaSQL* implementation that makes use of existing technology. We also discussed the implementation algorithms and related optimization opportunities.

Legacy as well as non-traditional information systems constitute an important fragment of the data sources available in real-life. We demonstrated that SchemaLog can be naturally extended to support non-relational systems as well. In particular, we showed that the powerful features of SchemaLog for meta-data querying, information restructuring, and interoperability extend naturally to the ER databases, Network databases, and information sources on the Web. The Web being an important and extensive source of non-traditional data, we gave it a special treatment in this thesis. In particular, we addressed the fundamental problem of retrieving specific information of interest to the user, from the enormous number of resources that are available on the Web. With this in mind, and inspired by SchemaLog, we developed a simple logic called *WebLog*. We presented a conceptual model for HTML documents and studied its syntax and (informal) semantics. One of the novelties of *WebLog* is that it provides a first class treatment of hyperlinks in HTML documents, and makes use of this to express powerful forms of navigation across HTML documents. We illustrated the simplicity and power of *WebLog* using a variety of applications involving real-life information in the Web. We also discussed the other relevant proposals for querying structured documents with the Web as a special case, and the features of *WebLog* that makes it a unique language for Web querying and restructuring.

SchemaLog's versatility to adapt to diverse data models establishes its role as a medium for developing advanced mediators capable of integrating information from heterogeneous data sources in a datamodel independent manner.

6.2 Future Work

In this section, we delineate the many opportunities for future research opened up by this thesis.

1. **Alternative semantics in the context of full-fledged SchemaLog programs:** The semantics of programs in a database programming language based on full-fledged SchemaLog, raises interesting issues. In such a language, SchemaLog molecules can appear in rule heads, and have the effect of 'creating' new federation objects during the execution of the program. We can associate a notion of various levels of semantics, based on how we would like to treat objects created during the execution of the program. These semantics range from the most conservative (in which the higher-order variables in the program range only over the originally existing, edb objects) to the most liberal (in which variables

range over existing objects as well as objects created during the execution of the program). Issues such as expressive power of the language and efficient implementability are directly related to the varying notions of semantics and warrant further investigation.

2. **Challenges arising in the implementation of SchemaLog:** [ALNI96] investigated the major issues in the implementation of a SchemaLog-based database programming language. It proposed an architecture for the implementation, based on compiling SchemaLog constructs into the schema algebra ($\mathcal{SA}$) (proposed in Chapter 3). This work addressed some of the challenging issues unique to SchemaLog implementation and proposed three alternative storage structures for dealing with them. It also proposed algorithms for top-down implementation of SchemaLog, including alternative strategies for the implementation of the algebraic operators and evaluated the effectiveness of the alternative strategies with a series of experiments on top of MS Access. This work showed that the $\mathcal{SA}$ algebraic operators can be used in efficient set-oriented top-down evaluation of SchemaLog programs, by organizing computations in such a way that results are computed "piecemeal". However, $\mathcal{SA}$ is not a sufficiently powerful language to express every program in the full-fledged SchemaLog language. Development of an algebra in the spirit of $\mathcal{SA}$ and having an equivalent expressive power as the full SchemaLog language is an interesting research issue.

   Another challenging problem in the context of realizing SchemaLog is the following. An important feature of SchemaLog that drives its powerful restructuring capabilities is the ability to 'program' using tuple-id's. However, in doing so, some perfectly natural SchemaLog programs have the effect of producing *identical tuples* in the output, albeit with different tuple-id's. This is clearly undesirable, since eliminating these redundant tuples is a costly operation. Avoiding the generation of duplicate tuples during the processing of SchemaLog programs is an interesting open research problem. [Ala97] addresses these and other related issues such as adapting the magic-sets query processing method to the context of processing SchemaLog programs.

   While the discussions above are based on realizing a non-intrusive implementation of SchemaLog based on existing technology, the inherently two-dimensional nature of SchemaLog suggests that, novel physical storage structures and and attendant multidimensional indexing technologies would lead to a more efficient implementation. Emerging technologies such as OLAP would benefit immensely from research in this direction.

3. **Extending *SchemaSQL* for OLAP Applications:** In Chapter 4, we outlined the possibility of *SchemaSQL* as a language for OLAP applications. However, *SchemaSQL* , as presented in this chapter, suffers from some important limitations in this context. Some of these limitations include *(1)* lack of any notion similar to typing, *(2)* limited dynamic schema creation capability (a view needs to be defined in order to restructure information, as opposed to being able to express restructuring *queries*), and *(3)* loss of information on meta-data (sometimes restructuring a relation using a *SchemaSQL* statement can result in loss of valuable information *about* meta-data). [Gin97] investigates these issues in depth and proposes extensions to the language in such a way that it can serve as a full-fledged language for OLAP applications.

   The strategy for implementing *SchemaSQL* discussed in Chapter 4 (Algorithm 4.5.1), considers only its querying fragment. It would be interesting to study how this strategy would scale up when restructuring is incorporated in our implementation. [Gin97] addresses these and related issues. [Urs97] discusses an efficient implementation of a related algebra (obtained by extending the tabular algebra of [GLS96]) for multidimensional databases.

4. **Query processing in the context of legacy systems:** In Chapter 5, we proposed two possibles strategies for realizing an implementation of SchemaLog on top of legacy systems such as the network and ER database systems: *(1)* based on materializing the relational abstractions of the underlying data sources, and *(2)* based on a direct translation of SchemaLog query into the host database DML. Both the strategies provide opportunities for interesting research. In particular, implementing the tid's of SchemaLog in *(1)*, and the optimization issues that arise in *(2)* in the context of compiling the SchemaLog constructs into those of the host language constructs, raise many challenges requiring further research.

5. **Semantic abstractions on Web sources:** The conceptual model underlying our extension to SchemaLog to the Web setting is based on the HTML grammar. Thus, our abstraction of Web documents is essentially *syntactic* - *all* documents have an identical mapping (based on the HTML grammar) to their corresponding abstractions; in particular, the abstractions do not take into account the *semantics* of the document. Though SchemaLog can be further used to build semantic abstractions on top of the syntactic view, a strategy for *directly* building semantic abstractions of HTML documents is clearly desirable. In the context of integrating data from heterogeneous semi-structured data sources, [LSGK96] takes the first step in this direction and proposes a methodology for building semantic views on top of data in applications such as spreadsheets, word processors, and mail tools. Based on such a view, these tools can be queried and updated in the same vein as that can be done in conventional database systems. We need to investigate how such a methodology can be extended to the Web setting. Challenging issues related to query optimization and (semantic) view maintenance arise in this context.

Yet another opportunity for future research in the context of *WebLog*, arises in the development of its formal semantics. Implementation of *WebLog* also poses many interesting problems. A restricted fragment of *WebLog* has been implemented ([KLS96]). We are currently in the process of realizing a full-blown implementation. In this context, we are investigating various useful foreign functions and their efficient incorporation in the *WebLog* engine. Our ongoing work addresses these and related issues.

Bibliography

[Abi97] Abiteboul, Serge. Querying Semi-Structured Data. In *6th International Conf. on Database Theory*, Delphi, Greece, January 1997.

[AC75] Astrahan, M.M. and Chamberlin, D.D. Implementation of a Structured English Query Language. *Communications of the ACM*, 18:580–587, 1975.

[ACHK94] Arens, Y., Chee, C.Y., Hsu, C.N., and Knoblock, C. Retrieving and integrating data from multiple information sources. *Intl Journal of Intelligent Cooperative Information Systems*, 2:2, 1994.

[ACM90] *ACM Computing Surveys*, 22(3), Sept 1990. Special issue on HDBS.

[ACM93] Abiteboul, S., Cluet, S., and Milo, T. Querying and updating the file. In *Proc. of the Conf on Very Large Databases (VLDB)*, 1993.

[ACM94] ACM. *ACM Transactions on Database Systems*, volume 19, June 1994.

[ACM95] Abiteboul, S., Cluet, S., and Milo, T. A database interface for file update. In *ACM SIGMOD Intl. Conf. on Management of Data*, 1995.

[ADK+91] Ahmed, R., DeSmedt, P., Kent, W., Ketabchi, M., Litwin, W., Rafii, A., and Shan, M.C. Pegasus: A system for seamless integration of heterogeneous information sources. In *IEEE COMPCON*, pages 128–135, 1991.

[AG87] Abiteboul, S. and Grumbach, S. COL: A Logic-Based Language for Complex Objects. In *Proc. of Workshop on Database Programming Languages*, pages 253–276, 1987.

[Ala97] Alanaoly, Andrews J. On Implementing SchemaLog: An Advanced Database Programming Language. Master's thesis, Department of Computer Science, Concordia University, Montreal, Quebec, Canada, 1997. (In preparation).

[ALNI96] Alanoly Andrews, Laks V.S. Lakshmanan, Nematollaah Shiri, and Iyer N. Subramanian. On Implementing SchemaLog: An Advanced Database Programming Language. In *Intl. Conf. on Information and Knowledge Management*, Baltimore, MD., November 1996.

[AMMT96] Atzeni, Paolo, Mecca, Giansalvatore, Merialdo, Paolo, and Tabet, Elena. Structures in the web. Technical report, DDS, Sezione Informatica, Universita di Roma Tre, 1996. Technical Report (Submitted for Publication).

[ART96] Asirelli, P., Renso, C., and Turini, F. Language extensions for semantic integration of deductive databases. In *Proc. Intl. Workshop on Logic in Databases (LID'96)*, pages 415–434, San Miniato, Italy, July 1996. Springer-Verlag, LNCS-1154.

[ASD+91] Ahmed, R., Smedt, P., Du, W., Kent, W., Ketabchi, A., and Litwin, W. The pegasus heterogeneous multidatabase system. *IEEE Computer*, December 1991.

[BC95] Berners-Lee, T. and Connolly, D. Hypertext Markup Language - 2.0 (Work in Progress), 1995. URL:http://www.w3.org/hypertext/WWW/MarkUp/MarkUp.html.

[Bee93] Beech, D. Collections of objects in SQL3. In *Proc. 19th VLDB Conf.*, 1993.

[BG92] Barsalou, T. and Gangopadhyay, D. An open framework for interoperation of multimodel multidatabase systems. In *IEEE Data Engg.*, 1992.

[BLN86] Batini, C., Lenzerini M., and Navathe S.B. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surveys*, pages 323–364, December 1986.

[BR86] Bancilhon F. and Ramakrishnan R. An amateur's introduction to recursive query-processing strategies. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 16–52, 1986.

[CAS94] Christophides, V., Cluet, S. Abiteboul, S., and Scholl, M. From structured documents to novel query facilities. In *ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[CCS95] Codd, E.F., Codd, S.B., and Salley C.T. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate, 1995. White paper - URL:http://www.arborsoft.com/papers/coddTOC.html.

[CD88] Cuppens, F. and Demolombe, R. Cooperative answering: a methodology to provide intelligent access to databases. In *Second Intl. conf. on Expert Database Systems*, 1988.

[CDRS86] Carey, M., DeWitt, D., Richardson, J., and Shekita, E. Object and file management in the exodus extensible database system. In *Proc. Intl. Conf. on Very Large Databases*, 1986.

[CGB95] Clifton, Chris, Garcia-Molina, Hector, and Bloom, David. Hyperfile: A data and query model for documents. *VLDB Journal*, 4:45–86, 1995.

[CGH+94] Chawathe, S., Garcia-Molina, H., Hammer, H., Ireland, K., Papakonstantinou, Y., Ullman, J.D., and Widom, J. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proc. of IPSJ*, Tokyo, Japan, 1994.

[Cha76] Chamberlin, D.D. et al. EQUEL'2: A Unified Approach to Data Definition, Manipulation, and Control. *IBM Journal of Research and Development*, 20:560–575, 1976.

[CHS+95] Carey M.J., Hass L.M., Schwarz P.M., Arya M., Cody W.F., Fagin R., Flickner A., Luniewski W., Niblack W., Petkovic D., Thomas J., Williams J.H., and Wimmers E.L. Towards Heterogeneous Multimedia Information Systems: the Garlic Approach. In *5th Intl. Workshop on Research Issues in Data Engineering (RIDE-DOM'95): Distributed Object Management*, pages 124–131, 1995.

[CKW89] Chen W., Kifer M., and Warren D.S. Hilog as a platform for database language. In *2nd Intl. Workshop on Database Programming Languages*, June 1989.

[CKW90] Chen W., Kifer M., and Warren D.S. A foundation for higher-order logic programming. Technical report, SUNY at Stony Brook, 1990. (Preliminary versions appear in Proc. 2nd Intl. Workshop on DBPL, 1989 and Proc. NACLP 1989.).

[CL73] Chang, C.L. and Lee, R.C.T. *Symbolic Logic and Mechanical Theorem Proving.* New York, Academic Press, 1973.

[CL93] Chomicki, J. and Litwin, W. Declarative definition of object-oriented multidatabase mappings. In Ozsu, M.T, Dayal, U, and Valduriez, P, editors, *Distributed Object Management.* M. Kaufmann Publishers, Los Altos, California, 1993.

[Cod79] Codd, E.F. Extending the database relational to capture more meaning. *ACM TODS*, 4:560–575, 1979.

[DH84] Dayal, U. and Hwang, H. View definition and generalization for database integration in a multi-database system. *IEEE Transactions on Software Engineering*, 10(6):628–644, 1984.

[Dob95] Dobbie, Gillian. Foundations of deductive object-oriented database systems. Phd dissertation, research report, University of Melbourne, Parkville, Australia, March 1995.

[DP90] Desai, B. C. and Pollock, R. MDAS: Multiple Schema Integration Approch. 13(2):16–21, 1990.

[End72] Enderton, Herbert B. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[GBLP96] Gray, J., Bosworth, A., Layman, A., and Pirahesh H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotals. In *Proc. of the 12th Intl. Conf. on Data Engineering*, pages 152–159, 1996.

[GD96] Gori, Mario and Della Lena, Fabio. A Schemalog Implementation for a Mediator Language. Master's thesis, Department of Computer Science, University of Pisa, Pisa, Italy, October 1996.

[GGM92] Gaasterland, T., Godfrey, P., and Minker, J. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1:123–157., 1992.

[Gin97] Gingras, Frédéric. Extending SchemaSQL Towards Multidimensional Databases and OLAP. Master's thesis, Department of Computer Science, Concordia University, Montreal, Quebec, Canada, 1997. (In preparation).

[GLL93] Grant, John, Ling, Tok, and Lee, Mong. ERL: Logic for Entity-Relationship Databases. *Journal of Intelligent Information Systems*, 2:115–147., 1993.

[GLRS93] John Grant, Witold Litwin, Nick Roussopoulos, and Timos Sellis. Query languages for relational multidatabases. *VLDB Journal*, 2(2):153–171, 1993.

[GLS95] Gyssens, Marc, Lakshmanan, L.V.S., and Subramanian, I. N. Tables as a paradigm for querying and restructuring. Tech. report, Concordia University, Montreal, November 1995.

[GLS96] Gyssens, Marc, Lakshmanan, L.V.S., and Subramanian, I. N. Tables as a paradigm for querying and restructuring. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, June 1996.

[Ham94] Hammer, J. *Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems*. PhD thesis, Computer Science Department, University of Southern California, Los Angeles, CA, 1994.

[HBP94] Hurson, A.R., Bright, M.W., and Pakzad, S. *Multidatabase Systems : An Advanced Solution For Global Information Sharing*. IEEE Computer Society, Los Alamitos, CA, 1994. Collection of Papers.

[HM93] Hammer, J. and McLeod, D. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *Intl. Journal of Intelligent & Cooperative Information Systems*, 2(1), 1993.

[Hsi92] Hsiao, D.K. Federated databases and systems: Part-one - a tutorial on their data sharing. *VLDB Journal*, 1:127–179, 1992.

[HVM95] Hasan, Masum, Vista, Dimitra, and Mendelzon, Alberto. Visual web surfing with hy+. In *CASCON*, 1995.

[KCGS93] Kim, W., Choi, I., Gala, S.K., and Scheevel, M. On resolving schematic heterogeneity in multidatabase systems. *Distributed and Parallel Databases*, 1(3), 1993.

[KGK$^+$95] Kelley, W., Gala, S. K., Kim, W., Reyes, T.C., and Graham. B. Schema architecture of the UniSQL/M multidatabase system. In *Modern Database Systems*. 1995.

[Kim90] Kim, Won. *Introduction to Object Oriented Databases*. MIT Press, 1990.

[KKS92] Kifer Michael, Kim Won, and Sagiv, Yehoshua. Querying object-oriented databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 393–402, 1992.

[KL88] Kifer M. and Li A. On the semantics of rule-based expert systems with uncertainty. In M. Gyssens, J. Paradaens, and D. van Gucht, editors, *2nd Intl. Conf. on Database Theory*, pages 102–117, Bruges, Belgium, August 31-September 2 1988. Springer-Verlag LNCS-326.

[KLK91] Krishnamurthy, R., Litwin, W., and Kent, W. Language features for interoperability of databases with schematic discrepancies. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 40–49, 1991.

[KLS96] Kegl, Balazs, Lakshmanan, L.V.S., and Subramanian, I. N. Realizing WebLog. Tech. report, Dept. of CS, Concordia Univ., Montreal, Canada, May 1996.

[KLW95] Kifer M., Lausen G., and Wu J. Logical foundations for object-oriented and frame-based languages. *Journal of ACM*, May 1995. (Tech. Rep., SUNY Stony Brook, 1990).

[KN88] Krishnamurthy, R. and Naqvi, S. Towards a real Horn clause language. In *Proc. 14th VLDB Conf.*, pages 252–263, 1988.

[Kos] Koster, M. World wide web wanderers, spiders, and robots. URL: http://web.nexor.co.uk/mak/doc /robots/robots.html.

[KS92] Kifer Michael and Subrahmanian V.S. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367, 1992.

[KS95] Konopnicki, David and Shmueli, Oded. W3qs : A query system for the world-wide web. In *Proc. 21st VLDB Conf*, pages 54–65, 1995.

[Law93] Lawley, M. J. A prolog interpreter for f-logic. Tech. report, Griffith University, 1993.

[LBT92] Lefebvre, A., Bernus, P., and Topor, R. Query transformation for accessing heterogeneous databases. In *Workshop on Deductive Databases in conjunction with JICSLP*, pages 31–40, November 1992.

[Lit89] Litwin. W. MSQL: A multidatabase language. *Information Scicncc*. 48(2), 1989.

[LMR90] Litwin. Witold. Mark, Leo, and Roussopoulos. Nick. Interoperability of multiple autonomous databases. *ACM computing surveys*, 22(3):267–293. Sept 1990.

[LN90] Lipton, Richard and Naughton, Jeffrey. Query size estimation by adaptive sampling. In *Proc. ACM PODS*, 1990.

[LNS90] Lipton, Richard, Naughton, Jeffrey, and Schneider, Donovan. Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD*, 1990.

[LR82] Landers, T. and Rosenberg, R. An overview of multibase. *Distributed Databases*, pages 153–184, 1982.

[LRO96] Levy, A.Y., Rajaraman, A., and Ordille, J.J. Querying heterogeneous information sources using source descriptions. In *Proc. 22nd VLDB Conf.*, pages 251–262, 1996.

[LS94] Lakshmanan Laks V.S. and Sadri F. Modeling uncertainty in deductive databases. In *Proc. Intl. Conf. on Database Expert Systems and Applications (DEXA '94)*, Athens, Greece, September 1994. Springer-Verlag, LNCS-856.

[LS95] Lakshmanan, Laks V.S. and Subramanian, Iyer N. On higher-order logics for multidatabase interoperability. Tech. report, Concordia University, Montreal, Quebec, 1995.

[LSGK96] Lakshmanan, Laks V.S., Subramanian, Iyer N., Goyal, Nita, and Krishnamurthy, Ravi. On querying and updating the spreadsheets. Technical report, Concordia University, Montreal, Quebec, October 1996. Technical Report.

[LSK95] Levy, A.Y., Srivastava, D., and Kirk, T. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 4. Sept 1995. Special Issue On Networked Information Systems.

[LSPS95] Lakshmanan L.V.S., Subramanian I. N., Papoulis Despina. and Shiri Nematollaah. A declarative system for multidatabase interoperability. In V. S. Alagar, editor, *Proc. of the 4th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST)*, Montreal, Canada, July 1995. Springer-Verlag. Tools Demo.

[LSS95] Lakshmanan L.V.S., Sadri F., and Subramanian, I. N. Extending database technology for sophisticated database programming. Tech. report, Concordia University, Montreal, June 1995.

[LSS96a] Lakshmanan L.V.S., Sadri F., and Subramanian, I. N. A declarative language for querying and restructuring the web. In *Proc. of the Sixth*

International Workshop on Research Issues on Data Engineering: Interoperability of Nontraditional Database Systems (RIDE-NDS'96), New Orleans, Louisiana, February 1996.

[LSS96b] Lakshmanan L.V.S., Sadri F., and Subramanian, I. N. SchemaSQL – a language for querying and restructuring multidatabase systems. Tech. report, Concordia University, Montreal, 1996. In Preparation.

[LSS97] Lakshmanan L.V.S., Sadri F., and Subramanian I. N. Logic and algebraic languages for interoperability in multidatabase systems. Technical report, Concordia University, Montreal, November 1997. To Appear in the Journal of Logic Programming Vol 33, No 2, pp. 101-149.

[Man89] Manchanda, S. Higher-order logic as a data model. In *Proc. of the North American Conf. on Logic Programming*, pages 330–341, 1989.

[McB94] McBryan, O.A. Genvl and wwww: Tools for taming the web. In *Proc. of the First Intl. WWW Conf.*, May 1994.

[MGE+95] Masum Hasan, Gene Golovchinsky, Emanuel Noik, Nipon Charoenkitkarn, Mark Chignell, Alberto Mendelzon, and David Modjeska. Browsing local and global information. In *CASCON*, 1995.

[ML94] Mauldin, M.L. and Leavitt, J.R. Web agent related research at the center for machine translation. August 1994. SIGNIDR Meeting, McLean, Virginia.

[MMM96] Mendelzon, Alberto, Mihaila, George, and Milo, Tova. Querying the World Wide Web. In *4th Intl. Conf. on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.

[MPR90] Mumick I.S., Pirahesh H., and Ramakrishnan R. The magic of duplicates and aggregates. In *Proc. 16th Intl. Conf. on Very Large Databases (VLDB'90)*, pages 264–277, Brisbane, Australia, 1990.

[MR93] Mumick, I.S. and Ross, K.A. Noodle: A language for declarative querying in object-oriented database. In *Proc. 3rd Intl. Conf. on Deductive and Object-Oriented Databases (DOOD '93)*, December 1993.

[MR95] Missier, P. and Rusinkiewicz, Marek. Extending a multidatabase manipulation language to resolve schema and data conflicts. In *Proc. Sixth IFIP TC-2 Working Conf. on Data Semantics (DS-6)*, Atlanta, May 1995.

[NR89] Nguyen G.T. and Rieu D. Schema evolution in object-oriented database systems. *Data and Knowledge Engg., North-Holland*, 4:43–67, 1989.

[Osb89] Osborn, Sylvia. The role of polymorphism in schema evolution in an object-oriented database. In *IEEE Trans. on Knowledge and Data Engg.*, pages 310–317, Sept 1989.

[Pap94] Papoulis, Despina. Realizing schemalog. Tech. report, Dept. of CS. Concordia Univ., Montreal, Canada, 1994.

[Poi95] Point Communications Corp. Lycos: the catalog of the internet, 1995. URL: http://www.pointcom.com/jpegs/reviews/6-28-031.htm.

[Ros92] Ross, Kenneth. Relations with relation names as arguments: Algebra and calculus. In *Proc. 11th ACM Symp. on PODS*, pages 346–353, June 1992.

[RSS92] Ramakrishnan R., Srivastava D., and Sudarshan S. CORAL: Control, relations, and logic. In *Proc. Intl. Conf. on Very Large Databases*, 1992.

[SAB$^+$95] Subrahmanian, V.S., Adali, S., Brink, A., Emery, R., Lu, J.J. Rajput, A., Rogers, T.J., Ross, R., and Ward, C. HERMES: Heterogeneous Reasoning and Mediator System. Tech. report, submitted for publication, Institute for Advanced Computer Studies and Department of Computer Science University of Maryland, College Park, MD 20742, 1995.

[Sad91] Sadri, Fereidoon. Modeling uncertainty in databases. In *Proc. 7th IEEE Intl. Conf. on Data Eng.*, pages 122–131, April 1991.

[SEKN92] Schwartz, M.F., Emtage, A., Kahle, B., and Neuman, B. C. A comparison of internet resource discovery approaches. *Computing Systems*, 5(4), 1992. URL: ftp://ftp.cs.colorado.edu/pub/cs/techreports /schwartz/PostScript/RD.Comparison.ps.Z.

[She88] Shepherdson, J.C. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1988.

[SHT+77] Shu, N. C., Housel, B.C., Taylor, R.W., Ghosh, S.P., and Lum, V.Y. Express: a data extraction, processing, and restructuring system. *ACM Transactions on Database Systems*, 2, 2, June 1977.

[Sig91] Semantic Issues in Multidatabase Systems. *Sigmod Record*, 20(4), December, 1991. Special Issue Edited by Amit Sheth.

[SL90] Sheth, Amit P. and Larson, James A. Federated database system for managing distributed, heterogeneous and autonomous databases. *ACM computing surveys*, 22(3):183–236, Sept. 1990.

[SQL96] SQL Standards Home Page. SQL 3 articles and publications, 1996. URL: www.jcc.com/sql_articles.html.

[SSR94] Sciore, E., Siegel, M., and Rosenthal, A. Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Transactions on Database Systems*, 19(2):254–290, June 1994.

[Sub87] Subrahmanian, V.S. On the semantics of quantitative logic programs. In *Proc. 4th IEEE Symposium on Logic Programming*, pages 173–182. Computer Society Press, Washington DC, 1987.

[Sub94] Subrahmanian, V.S. Amalgamating knowledge bases. *ACM Transactions on Database Systems*, 19, 2:291–331, 1994.

[Tem87] Templeton, M., *et al.* Mermaid: A front-end to distributed heterogeneous databases. In *Proc. IEEE 75, 5*, pages 695–708, May 1987.

[TRV96] Tomasic, A., Raschid, L., and Valduriez, P. Scaling heterogeneous databases and the design of disco. In *Proc. IEEE Intl. Conf. on Distributed Computing Systems*, 1996.

[Ull84] Ullman, J.D. *Principles of Database Systems*, volume I. Computer Science Press, Maryland, 1984.

[Ull87] Ullman, J.D. Database theory: Past and future. In *Proc. of the ACM Symp. PODS*, 1987.

[Ull89a] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, Maryland, 1989.

[Ull89b] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, Maryland, 1989.

[Urs97] Ursu, Ioana M. On Efficiently Implementing the Tabular Data Model. Master's thesis, Department of Computer Science, Concordia University, Montreal, Quebec, Canada, 1997. (In preparation).

[vK76] van Emden, M.H. and Kowalski, R.A. The semantics of predicate logic as a programming language. *JACM*, 23(4):733–742, October 1976.

[Wie92] Wiederhold, G. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, March 1992.

[WK79] Wong, Eugene and Katz, Randy H. Logical Design and Schema Conversion for Relational and DBTG Databases. In P.P. Chen (ed.), editor, *Entity-Relationship Approach to System Analysis and Design*, pages 311–321. North-Holland Publishing Company, 1979.

Appendix A

SchemaLog

A.1 Equality

For simplicity of exposition, we have left open the issue of how equality is to be interpreted, in our presentation of model theory and proof theory. A straightforward approach is to view equality semantically. For instance, if a Herbrand structure $\mathcal{H}$ contains both the atoms $d :: r[i : a{\rightarrow}v_1]$ and $d :: r[i : a{\rightarrow}v_2]$, then we can force $\mathcal{H}$ also to contain the atom $v_1 \doteq v_2$, which says the terms v_1 and v_2 semantically denote the same intension. The idea then is to consider the quotient Herbrand structures with respect to the congruence $\doteq$. The proof theory can be correspondingly augmented with paramodulation while preserving the soundness and completeness theorems. F-logic [KLW95] follows this approach. While there are some advantages to this approach, we feel that from a practical perspective on database querying, it is more natural to view equality syntactically. For example, if we have both $d :: emp[i : sal{\rightarrow}50K]$ *and* $d :: emp[i : sal{\rightarrow}100K]$ it is more appropriate to conclude our knowledge is *inconsistent* than to regard $50K$ and $100K$ as "semantically equal". The following definition of *e-satisfiability* formalizes the notion of syntactic equality.

Definition A.1.1 *A theory $\mathcal{T}$ is* e-satisfiable *if it has a model such that distinct ground terms in the language of $\mathcal{T}$ are interpreted by the model into different intensions.*

Corresponding to the model-theoretic property of e-satisfiability, we introduce its proof-theoretic counterpart – *e-consistency*.

Definition A.1.2 *Let A be the atom $db :: rel[t : a \rightarrow v]$ and B be $db' :: rel'[t' : a' \rightarrow v']$. A and B are* e-ambivalent *if there are ground substitutions θ and θ' such that $\langle db, rel, t, a\rangle\theta \equiv \langle db', rel', t', a'\rangle\theta'$ and $v\theta \not\equiv v'\theta'$.*

A theory $\mathcal{T}$ is e-inconsistent *if there exist e-ambivalent atoms A and B (not necessarily distinct) such that $\mathcal{T} \vdash A$ and $\mathcal{T} \vdash B$. $\mathcal{T}$ is* e-consistent *if $\mathcal{T}$ is not e-inconsistent.*

Note that a single atom could be e-ambivalent with itself. E.g. consider the theory $\mathcal{T} = \{d :: r[i : a \rightarrow X]\}$.

We next lift the soundness and completeness theorem (Theorem 2.5.4) of Section 2.5, to account for e-satisfiability and e-consistency.

Theorem A.1.1 *A theory $\mathcal{T}$ is e-consistent iff $\mathcal{T}$ is e-satisfiable.*

PROOF. ($\Rightarrow$) $\mathcal{T}$ is e-consistent. Assume $\mathcal{T}$ is *not* e-satisfiable. Then there exist ground atoms $A \equiv d :: r[i : a \rightarrow v_1]$ and $B \equiv d :: r[i : a \rightarrow v_2]$, v_1 and v_2 are distinct, such that $\mathcal{T} \models A$ and $\mathcal{T} \models B$. Clearly A and B are *e-ambivalent*. By Theorem 2.5.4, $\mathcal{T} \vdash A$ and $\mathcal{T} \vdash B$, which implies $\mathcal{T}$ is e-inconsistent – a contradiction!

($\Leftarrow$) $\mathcal{T}$ is e-satisfiable. Assume $\mathcal{T}$ is e-inconsistent. There exist e-ambivalent atoms A and B such that $\mathcal{T} \vdash A$ and $\mathcal{T} \vdash B$. Let θ and θ' be substitutions such that $A\theta$ and $B\theta'$ are ground atoms that agree on all the components except the value component. By Theorem 2.5.4, $\mathcal{T} \models A\theta$ and $\mathcal{T} \models B\theta'$. It follows that $\mathcal{T}$ is not e-satisfiable - a contradiction! □

A.2 Proofs of Some Results

Theorem 2.5.2. (*Herbrand's Theorem*) *A set S of wffs in clausal form is unsatisfiable iff every complete semantic tree T for S has a finite closed subtree.*

PROOF. It has been shown in Section 2.4 that there is a transformation from SchemaLog to first order logic such that a SchemaLog formula $\mathcal{A}$ is true in a structure M_s under vaf ν iff the corresponding first order formula encode($\mathcal{A}$) is true in the corresponding first order structure encode(M_s) under the vaf ν (Theorem 2.4.1). Herbrand's theorem can now be proved from the above result using a technique similar to that used for predicate calculus [CL73]. The main observation is that whenever S is unsatisfiable, every branch of any complete semantic tree T of S must have a failure node. Since each node of T has a finite number of children, an application of König's Lemma at once implies the existence of a finite closed subtree of T. The details are straightforward and are suppressed. □

Lemma 2.5.4. *(Lifting Lemma) If C_1' and C_2' are instances of C_1 and C_2, respectively, and if C' is a resolvent of C_1' and C_2', then there is a resolvent C of C_1 and C_2 such that C' is an instance of C.*

PROOF. Variables in C_1 and C_2 can be renamed such that there are no common variables in them. Let L_1' and L_2' be the literals of C_1' and C_2' (respectively) that are resolved upon and let γ be the mgu of L_1' to $\neg L_2'$. Let C' be the clause obtained by removing $L_1'\gamma$ and $L_2'\gamma$ from a disjunction of $C_1'\gamma$ and $C_2'\gamma$. There is a substitution θ such that $C_1' = C_1\theta$ and $C_2' = C_2\theta$. Let λ_i be the mgu for the literals, say $\{L_i^1, \ldots, L_i^{k_i}\}$ in C_i, which correspond to L_i'. Let $L_i \equiv L_i^1\lambda_i \equiv \cdots \equiv L_i^{k_i}\lambda_i$. Clearly, L_i is a literal in the factor $C_i\lambda_i$ of C_i. It follows from this that L_i' is an instance of L_i. Since L_1' is unifiable to $\neg L_2'$, L_1 is unifiable to $\neg L_2$. Let σ be the mgu of L_1 to $\neg L_2$.

Let C be the disjunction $D_1 \vee D_2$ where D_i is the disjunction obtained by removing $L_i\sigma$ from $(C_i\lambda)\sigma$, $i = 1, 2$. From this, it can be proved that C is a resolvent of C_1 and C_2. Clearly, C' is an instance of C since $C' = E_1 \vee E_2$, where E_i is obtained by removing $L_i'\gamma$ from $(C_i'\gamma)\sigma$, $i = 1, 2$, and $\lambda \circ \sigma$ is more general than $\theta \circ \gamma$. □

Appendix B

Some Direct Results on the Expressive Powers of $\mathcal{SQA}$ and $\mathcal{L}_C$

The following lemma establishes that the expressive power of $\mathcal{SQA}$ is no less than that of safe $\mathcal{L}_C$ and thus (in conjunction with Lemma 3.5.1) establishes that the two languages have the same expressive power.

Lemma B.0.1 *Every safe $\mathcal{L}_C$ query can be expressed by a $\mathcal{SQA}$ expression.*

PROOF. This proof is similar to the proof for the classical case as discussed in [Ull84]. There are two major parts to this proof. In the first part, we prove that given a safe $\mathcal{L}_C$ formula ψ, there is an expression in $\mathcal{SQA}$, denoting the set $DOM(\psi)$. The second part is an inductive proof on the number of operators in a subformula of ψ.

Let $\{x_1, \ldots, x_k \mid \psi(x_1, \ldots, x_k)\}$ be a safe $\mathcal{L}_C$ formula. It can be shown that for every safe $\mathcal{L}_C$ formula, there exists an equivalent safe formula with no occurrences of $\wedge$ and $\forall$. Thus, we may assume without loss of generality that ψ has only operators $\vee$, $\neg$, and $\exists$.

Part I: We need to prove that $\mathcal{SQA}$ can generate DOM, the set of all constants appearing in ψ and in the databases in the federation. As our framework treats attribute names and relation names as first class citizens, the $\mathcal{SQA}$ expression generating DOM should include them in the domain. If C is the set of all constants appearing in $\mathcal{P}$, DOM is expressed the following way.

$$DOM = C \cup \delta() \ \cup \pi_2(\rho(\delta())) \ \cup \pi_3(\alpha(\rho(\delta()))) \ \cup \pi_4(\gamma_{\langle \ \rightarrow \ \rangle}(\rho(\delta())))$$

Part II: We prove by induction on the number of operators in a subformula ω of ψ that, if ω has free domain variables $y_1, \ldots, y_m$, then $DOM(\psi)^m \cap$

$\{y_1, \ldots, y_m \mid \omega(y_1, \ldots, y_m)\}$ has an equivalent expression in $\mathcal{SQA}$. Thus, as a special case, when ω is ψ itself, we have an algebraic expression for $DOM(\psi)^k \cap \{x_1, \ldots, x_k \mid \psi(x_1, \ldots, x_k)\}$, and we would have proved the theorem. The inductive proof is similar to the one for the classical case ([Ull84]) - the base case is when there are zero operators in ω and the other cases are for operators $\vee$, $\neg$, and $\exists$. The only difference in our setting is the base case which involves the new atoms in $\mathcal{L}_C$. We show how relations corresponding to $\mathcal{L}_C$ atoms can be derived using $\mathcal{SQA}$. There are four scenarios to consider depending on the depth of ω.

(a) Let ω be X. Then $E_i = \boldsymbol{\delta}()$. If ω is a constant d, then E_i is simply $\{d\}$.

(b) Let ω be $D :: R$. Then $E_i = \boldsymbol{\rho}(\boldsymbol{\delta}())$. If one or more of D, R are constants, or if D and R are the same variable, then simply modify E_i by imposing appropriate additional selection(s).

(c) If ω is $D :: R[A_1, \ldots, A_n]$, then E_i is essentially the expression $\boldsymbol{\alpha}(\boldsymbol{\rho}(\boldsymbol{\delta}()))$ θ-joined with itself n-times, where θ is '$\$1 = \$1 \wedge \$2 = \$2$'. If some of the terms in ω are constants or repeating variables, we can impose appropriate selections in E_i.

(d) If ω is of the form $D :: R[A_1 \rightarrow V_1, A_2 \rightarrow V_2, \ldots, A_n \rightarrow V_n]$, then E_i is $\boldsymbol{\pi}_{outputArgs}(\sigma_{conditions}\gamma_{\langle p_1 \ldots p_n \rangle}(\boldsymbol{\rho}(\boldsymbol{\delta}())))$, where p_i is an attribute/value pair of one of the forms '$\rightarrow$', '$a_i \rightarrow$', '$\rightarrow v_i$', '$a_i \rightarrow v_i$', depending on whether and where the pair $A_i \rightarrow V_i$ contains constants. $\sigma_{conditions}$ corresponds to selection conditions capturing the occurrence of constants and repeating variables in ω, and $outputArgs$ is the list of arguments corresponding to distinct variables occurring in ω.

Now the techniques of [Ull84] can be applied to obtain an expression for ψ. □

The following result establishes that the expressive power of db-safe $\mathcal{L}_C$ is no more that of d-$\mathcal{SQA}$.

Lemma B.0.2 *Every db-safe $\mathcal{L}_C$ query is expressible in d-$\mathcal{SQA}$.*

PROOF. This proof can also be obtained from the proof for the similar lemma (Lemma B.0.1) for the more liberal fragment of the language. In part I of the lemma, we need to prove that $DOM_{db}(\phi)$ can be generated using a d-$\mathcal{SQA}$ expression. The only modification in Part II arises because of the need for considering cases when variables appear in the database positions of db-safe $\mathcal{L}_C$ formulas.

Part I: Let $\{x_1, \ldots, x_k \mid \psi(x_1, \ldots, x_k)\}$ be a db-safe $\mathcal{L}_C$ query. Let C be the set of constants in ψ. We need to prove that $DOM_{db}(\psi)$ can be generated using a *d-$\mathcal{SQA}$* expression. Definition 3.5.5 defines DOM_{db} as a union of a sequence of sets, in which an element of the sequence is defined based on its predecessor, and the length of the sequence is based on the instance of the federation. Thus, it would appear that a corresponding algebraic expression that generates DOM_{db} would need an unbounded union operation (simulable using a *while* construct). However, it can be observed that in order to generate DOM_{db}, *it is enough that the length of the sequence be utmost the number of $\mathcal{L}_C$ atoms in ψ*. Thus, the union is indeed bounded and the number of unions in the equivalent *d-$\mathcal{SQA}$* expression, can be statically determined from ψ. Based on this crucial observation, we present the expression that generates $DOM_{db}(\psi)$.

$$DOM^0_{db}(\psi) = C.$$
$$DOM^{i+1}_{db}(\psi) = \hat{\delta}(DOM^i_{db}(\psi)) \cup \pi_2(\rho(\hat{\delta}(DOM^i_{db}(\psi)))) \cup \pi_3(\alpha(\rho(\hat{\delta}(DOM^i_{db}(\psi))))) \cup \pi_4(\gamma_{\langle \rightarrow \rangle}(\rho(\hat{\delta}(DOM^i_{db}(\psi)))))$$

$DOM_{db}(\psi) = \bigcup_{0 \leq i \leq n} DOM^i_{db}(\psi)$, where n is the number of $\mathcal{L}_C$ atoms appearing in ψ.

Part II: The major modification to the proof of this part from the proof of Lemma B.0.1 is the case when a variable appears in the database position of a subformula ω. Obtaining the *d-$\mathcal{SQA}$* expression when the database position contains a constant is straight-forward - these constants would be the elements of the argument relation s of $\hat{\delta}$. We illustrate the proof for the variable occurrence scenario for the case when the depth of ω is 1. Proofs for the other cases are similar.

Let ω be X. It follows from the definition of db-safety that X already occurs in another subformula τ of ψ. Let E_j be the *d-$\mathcal{SQA}$* expression corresponding to the db-safe formula $\{X \mid \tau(X)\}$. Then, $E_i = \hat{\delta}(E_j)$.

With this modification, the proof is obtained along the lines of the proof of Lemma B.0.1. ❑

We now present similar results for the more restrictive fragments of the languages.

Lemma B.0.3 *Every rel-safe $\mathcal{L}_C$ query can be expressed by a r-$\mathcal{SQA}$ expression.*

Lemma B.0.4 *Every attr-safe $\mathcal{L}_C$ query can be expressed by a a-$\mathcal{SQA}$ expression.*

Lemmas B.0.3 and B.0.4 can be proved using proof techniques akin to that of Lemma B.0.2 extended to the *rel* and *attr* fragments.

Appendix C

Extended-SchemaLog – Syntax and Semantics

In this appendix, we formally present the syntax and semantics of our language for querying and restructuring ER databases.

C.1 Syntax

We use strings starting with a lower case letter for constants and those starting with an upper case letter for variables. As a special case, we use t_i to denote arbitrary terms of the language. $\mathcal{A}, \mathcal{B}, \ldots$ denote arbitrary well-formed formulas and A, B, ... denote arbitrary atoms.

The vocabulary of the language $\mathcal{L}$ of SchemaLog consists of pairwise disjoint countable sets $\mathcal{G}$ (of function symbols), $\mathcal{S}$ (of non-function symbols), $\mathcal{V}$ (of variables), and the usual logical connectives $\neg, \vee, \wedge, \exists$, and $\forall$.

Every symbol in $\mathcal{S} \cup \mathcal{V}$ is a term of the language. If $f \in \mathcal{G}$ is a n-place function symbol, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

An *atomic formula* of $\mathcal{L}$ is an expression of one of the following forms:

<db>::<rel>[<tid>: <attr> $\rightarrow$ <val>]

<db>::<rel>[<attr>]

<db>::<rel>

<db>

<rel>[<tid>: <attr> $\rightarrow$ <val>]

<rel>[<attr>]
where <db>, <rel>, <attr>, <tid>, and <val> are terms of $\mathcal{L}$.

We refer to the first four categories of atoms as *db-atoms* and the last two as *rel-atoms.* In an atom of the form <db>::<rel>[<tid>: <attr> $\rightarrow$ <val>], we refer to the terms <db>, <rel>, <attr>, and <val> as the *non-id components* and <tid> as the *id component.* The id component intuitively stands for tuple-id (tid). The *depth* of an atomic formula A, denoted *depth*(A), is the number of non-id components in A. The depths of the four categories of db-atoms introduced above are 4,3,2, and 1 respectively. The well-formed formulas (wff's) of $\mathcal{L}$ are defined as usual: every atom is a wff; $\neg\mathcal{A}, \mathcal{A} \vee \mathcal{B}, \mathcal{A} \wedge \mathcal{B}, (\exists X)\mathcal{A}$, and $(\forall X)\mathcal{A}$ are wff's of $\mathcal{L}$ whenever $\mathcal{A}$ and $\mathcal{B}$ are wff's and X is a variable.

We also permit *molecular formulas* of the form
<db>::<rel>[<tid>: <attr$_1$> $\rightarrow$ <val$_1$>,...,<attr$_n$> $\rightarrow$ <val$_n$>] as an abbreviation of the corresponding well-formed formula
<db>::<rel>[<tid>: <attr$_1$> $\rightarrow$ <val$_1$>]$\wedge \ldots \wedge$ <db>::<rel>[<tid>: <attr$_n$> $\rightarrow$ <val$_n$>
A *literal* is an atom or the negation of an atom. A *clause* is a formula of the form
$\forall X_1 \ldots \forall X_m$
$(L_1 \vee \ldots \vee L_n)$ where each L_i is a literal and $X_1, \ldots, X_m$ are all the variables occurring in $L_1 \vee \ldots \vee L_n$. A *definite clause* is a clause in which at most one positive literal is present and it is represented as $A \leftarrow B_1, \ldots, B_n$ where A is called the *head* and $B_1, \ldots, B_n$ is called the *body* of the definite clause. A *unit clause* is a clause of the form $A \leftarrow$, that is a definite clause with an empty body.

C.2 Semantics

Let U be a non-empty set of elements called *intensions* (corresponding to the terms of $\mathcal{L}$). Consider a function $\mathcal{I}$ that maps each non-function symbol to its corresponding intension in U and a function $\mathcal{I}_{fun}$ which interprets the function symbols as functions over U. The true atoms of the model are captured using the functions $\mathcal{F}$ and $\mathcal{F}_{rel}$. $\mathcal{F}$ takes as arguments the name of the database, the relation name, attribute name, and tuple-id, and maps to a corresponding individual value. $\mathcal{F}_{rel}$ takes as arguments the name of a relation, attribute name, and tuple-id, and maps to a corresponding individual value. Thus for a given db (rel) atomic formula to be true, the function

$\mathcal{F}$ ($\mathcal{F}_{rel}$) corresponding to the formula (after mapping the symbols of the formula to their corresponding intensions) should be *defined* in the structure (and the values should match).

A *semantic structure* M for our language is a tuple $< U, \mathcal{I}, \mathcal{I}_{fun}, \mathcal{F}, \mathcal{F}_{rel} >$ where

- U is a non-empty set of intensions;
- $\mathcal{I}$: $\mathcal{S} \rightarrow U$ is a function that associates an element of U with each symbol in $\mathcal{S}$;
- $\mathcal{I}_{fun}(f) : U^n \rightarrow U$. where f is a function symbol of arity n in $\mathcal{G}$.
- $\mathcal{F} : U \rightsquigarrow [U \rightsquigarrow [U \rightsquigarrow [U \rightsquigarrow U]]]$, where $[A \rightsquigarrow B]$ denotes the set of all partial functions from A to B.
- $\mathcal{F}_{rel} : U \rightsquigarrow [U \rightsquigarrow [U \rightsquigarrow U]]$.

A *vaf* (variable assignment function) is a function $\nu : \mathcal{V} \longrightarrow U$. We extend it to the set $\mathcal{T}$ of terms as follows.

- $\nu(s) = \mathcal{I}(s)$ for every $s \in \mathcal{S}$,
- $\nu(f(t_1, ..., t_k)) = \mathcal{I}_{fun}(f)(\nu(t_1), ..., \nu(t_k))$, where f is a function symbol of arity k in $\mathcal{G}$ and t_i are terms.

Let $t_i \in \mathcal{T}$ be any term. The *satisfaction* of an atomic formula A, in a structure M under a vaf ν is defined as follows.

- Let A be of the form $t_1 :: t_2[t_4 : \; t_3 \rightarrow t_5]$. Then $M \models_\nu A$ iff
  $\mathcal{F}(\nu(t_1))(\nu(t_2))(\nu(t_3))(\nu(t_4))$ is defined in M, and
  $\mathcal{F}(\nu(t_1))(\nu(t_2))(\nu(t_3))(\nu(t_4)) = \nu(t_5)$

- Let A be of the form $t_1 :: t_2[t_3]$. Then $M \models_\nu A$ iff
  $\mathcal{F}(\nu(t_1))(\nu(t_2))(\nu(t_3))$ is defined in M.

- Let A be of the form $t_1 :: t_2$. Then $M \models_\nu A$ iff
  $\mathcal{F}(\nu(t_1))(\nu(t_2))$ is defined in M.

- Let A be of the form t_1. Then $M \models_\nu A$ iff
  $\mathcal{F}(\nu(t_1))$ is defined in M.

- Let A be of the form $t_1[t_3 : \; t_2 \rightarrow t_4]$. Then $M \models_\nu A$ iff
  $\mathcal{F}_{rel}(\nu(t_1))(\nu(t_2))(\nu(t_3))$ is defined in M, and
  $\mathcal{F}_{rel}(\nu(t_1))(\nu(t_2))(\nu(t_3)) = \nu(t_4)$

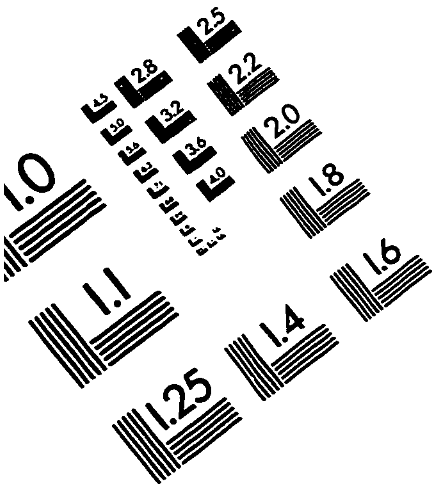
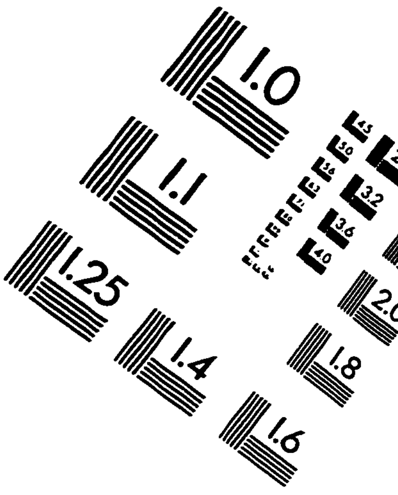
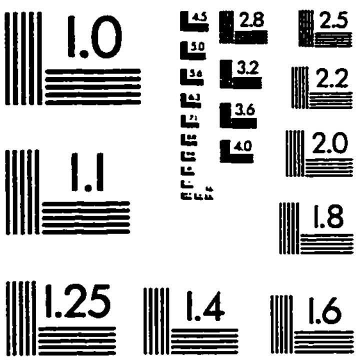
- Let A be of the form $t_1[t_2]$. Then $M \models_\nu A$ iff
  $\mathcal{F}_{rel}(\nu(t_1))(\nu(t_2))$ is defined in M.

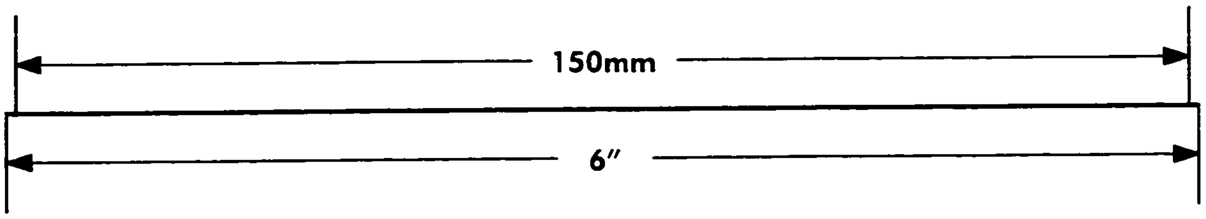
Satisfaction of compound formulas is defined in the usual way:

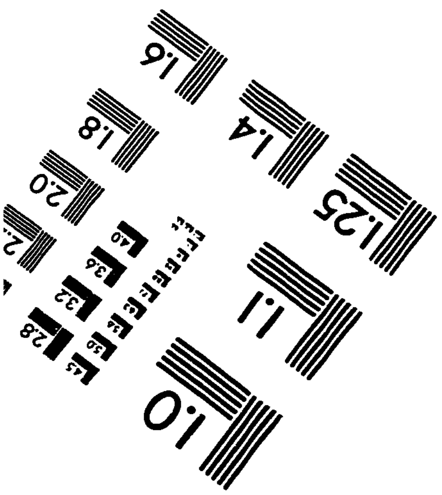
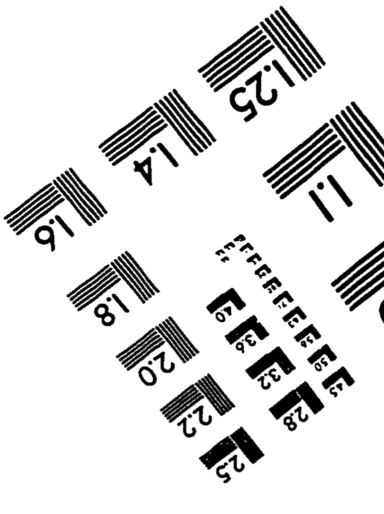
- $M \models_\nu (\mathcal{A} \vee \mathcal{B})$ iff $M \models_\nu \mathcal{A}$ or $M \models_\nu \mathcal{B}$;
- $M \models_\nu (\neg\mathcal{A})$ iff $M \not\models_\nu \mathcal{A}$;
- $M \models_\nu (\exists X)\mathcal{A}$ iff for *some* vaf μ, that may possibly differ from ν only on X, $M \models_\mu \mathcal{A}$;

For *closed* formulas, $M \models_\nu \mathcal{A}$ does not depend on ν and we can simply write $M \models \mathcal{A}$.

IMAGE EVALUATION
TEST TARGET (QA-3)



APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989