

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Exporting and utilizing database interfaces on the web

Permalink

<https://escholarship.org/uc/item/2xm3w3c0>

Author

Petropoulos, Michail

Publication Date

2006

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Exporting and Utilizing Database Interfaces on the Web

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Michail Petropoulos

Committee in charge:

Professor Yannis Papakonstantinou, Chair

Professor Alin Deutsch

Professor Dimitris N. Politis

Professor Vassilis J. Tsotras

Professor Victor Vianu

Professor Geoffrey M. Voelker

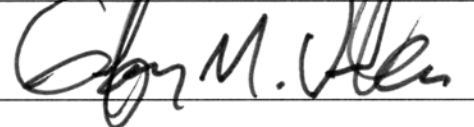
2006

Copyright
Michail Petropoulos, 2006
All rights reserved.

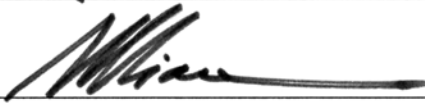
The dissertation of Michail Petropoulos is approved, and it is acceptable in quality and form for publication on microfilm:









 U. VIANU

 YANNIS PAPAKONSTANTINOU
Chair

University of California, San Diego

2006

Dedicated to my father, Athanasios Petropoulos

TABLE OF CONTENTS

	Signature Page	iii
	Dedication	iv
	Table of Contents	v
	List of Figures	viii
	Acknowledgements	x
	Vita, Publications, and Fields of Study	xii
	Abstract	xiii
I	Introduction	1
	A. Interactive Query Formulation Interfaces	4
	B. Data-Oriented Web Service Interfaces	8
	C. Web-based Query Form and Report Interfaces	10
	D. Thesis Overview	13
II	Interactive Query Formulation Interfaces	15
	A. Background	15
	1. Contributions	20
	B. Definitions and Notations	22
	C. Query Building Interfaces	23
	D. CLIDE Interaction in the Presence of Limited Access Methods	25
	1. Specification of CLIDE’s Color Scheme	29
III	The CLIDE Back-End	33
	A. Architecture	33
	B. Closest Feasible Queries Algorithm	35
	C. Color Algorithm	39
	D. Parameters	41
	E. Implementation	44
	F. Experimental Evaluation	48
	G. Discussion and Related Work	52
	H. Proofs	53
IV	Data-Oriented Web Service Interfaces	55
	A. Background	55
	1. Example	57
	B. Specifying Queries and Query Sets	59
	1. QSSL Specifications	59

2.	Reasoning about Data Services	63
3.	WSDL and XML Syntax	64
C.	QSSL Extensions	66
D.	Related Work	66
V	Graphical Query Interfaces for Semistructured Data	69
A.	Background	69
B.	System Overview and Architecture	71
C.	Related Work	73
1.	Novel Contributions of QURSED	79
D.	Data Model, XML Schema and Expanded Schema Tree	81
1.	Aliasing and <i>EST</i> Expansion	84
E.	Example <i>QFR</i> and End-User Experience	85
VI	Tree Query Language and Query Set Specifications	87
A.	Tree Query Language (TQL)	87
1.	Condition Tree	88
2.	Result Tree	93
B.	Query Set Specification	95
C.	Query Formulation Process	98
D.	Dependencies	101
VII	Editing Query Set Specifications	105
A.	Architecture	105
B.	Building Condition Tree Generators	107
1.	Automatic Introduction of Structural Disjunction	108
2.	Eliminating Redundancies	112
C.	Building Dependencies	114
D.	Building Result Tree Generators	115
1.	Schema-Driven Construction of Result Tree Generator	115
2.	Template-Driven Construction of Result Tree Generator	120
E.	Building Result Boolean Expressions	123
F.	Dynamic Projection Functionality	124
VIII	Conclusions and Future Work	127
A.	Conclusions	127
B.	Future Work	129
A	WSDL Specification of a Data Service	131
B	QSSX Syntax	133
C	Result XML Schema	138
D	TPX Query	140

E	XML Schema for TPX Syntax	142
F	TQL2XQuery Algorithm	145
G	GROUPBY Proposal	152
	Bibliography	154

LIST OF FIGURES

I.1	Data Integration and Publishing Architecture	2
I.2	Thesis Contributions	3
I.3	CLIDE Front-End	5
I.4	Data Service Architecture	10
I.5	Example <i>QFR</i> Interface	11
I.6	The QURSED Editor	12
II.1	Service-Oriented Architecture	16
II.2	Source Schemas and Web Services	18
II.3	QBE-Like Query Building Interfaces	24
II.4	Snapshots of an Interaction Session	27
II.5	Part of an Interaction Graph	30
III.1	CLIDE Architecture	35
III.2	MiniCon Optimizations and Extensions	45
III.3	CLIDE’s response time	51
IV.1	Data Service Architecture	57
IV.2	Airline Example	58
IV.3	Example Derivation	62
IV.4	Family Tree Recursive Example	63
V.1	The QURSED System Architecture	71
V.2	Example Data Set, XML Schema and Expanded Schema Tree	82
V.3	Example <i>QFR</i> Interface	85
VI.1	TQL Query Corresponding to Figure V.3	88
VI.2	Conjunctive Condition Trees	90
VI.3	OR-Removal Replacement Rules	90
VI.4	Resulting <i>loto</i> for Bindings of Table VI.1	95
VI.5	Query Set Specification	96
VI.6	Query Formulation Process	99
VI.7	Condition Tree Generator and Dependencies Graph	101
VI.8	Dependencies on the Query Form Page	102
VII.1	QURSED Editor Architecture	106
VII.2	Building a Condition Fragment	107
VII.3	“OR Node Introduction” Rules	111
VII.4	Example of the Construct <i>CTG</i> Algorithm	111
VII.5	“Node Elimination” Rule	113
VII.6	Eliminating Redundant Nodes on the <i>CTG</i>	113
VII.7	Building Dependencies	114
VII.8	Schema-Driven Constructed Report Page	115

VII.9	Selecting Elements Nodes and Constructing Template Report Page	116
VII.10	Automatically Generated Result Fragment, RTG and Template Report Page	117
VII.11	“OR Node Introduction” Rules for Result Fragment f_R	120
VII.12	Editing the Template Report Page	120
VII.13	Performing Element and Group-By Mappings on the Template Report Page	121
VII.14	Boolean Expressions for Dynamic Projection	125

ACKNOWLEDGEMENTS

The people acknowledged below had significant impact on the work presented in this thesis and on the way I evolved as a person during my PhD studies. They carry unique qualities that I try to inherit and pass to others. I could never imagine the wealth of ideas and visions they exposed me to when I started this journey. I am grateful for their generosity.

First and foremost, I would like to thank my advisor, Yannis Papakonstantinou, for giving me the opportunity to experience so many aspects of academic and professional life. From research, presentation skills and teaching at UCSD, to systems design, collaborations and business plans during the Enosys years, I enjoyed every bit of it. Not to mention our lengthy discussions on politics, history, music, movies and, of course, soccer. Moreover, he brought me in contact with other top researchers, who contributed in my progress and helped broaden my perspective of database research.

I thank all my coauthors for their patience and the time they devoted to teach me how research is conducted. First, Vasilis Vassalos, for helping me in my first professional and academic steps and for being a good friend. Alin Deutsch, for his unparalleled enthusiasm, clarity of mind and knowledge depth, and for his encouragement during critical times. Vagelis Hristidis, for being an inspirational colleague, and for listening to my complaints first and then making fun of them. Yannis Katsis, for his insightful comments and analytical skills that have been of great value to our projects. Also, I would like to thank Victor Vianu and all students of the database group at UCSD, for creating such a stimulating research environment.

Phil Bernstein's support has been of great importance to me on many different levels. I thank him for giving me the opportunity to experience the environment of Microsoft Research and meet many important people, for his contribution during the job search period and, finally, for his support during a difficult period of my life. Also, I would like to thank Juliana Freire and Prasan Roy for

our fruitful collaboration during my internship at Bell Labs, and Jussi Myllymaki, Paul Brown and Berthold Reinwald for my internship at IBM Almaden. Finally, the support from NSF and the San Diego Supercomputer Center during my years at UCSD is gratefully acknowledged.

Special thanks go to my friends Yiota Bafaki and Vasilis Stathopoulos for supporting me all these years from 10,000 miles away. To Dimitris Giannakos, for enabling this journey, for showing me the benefits of persistence, for helping me break the deadlock many years ago when nobody else could, and for adding so many dimensions to my life. I will keep reminding him of his impact in the years to come. To my mother Evaggelia and my sister Xenia, for their unconditional support and for filling my life with joy. And to my late father, for never giving up, even under extreme circumstances, always aiming for the impossible and for sacrificing personal benefit for the benefit of others.

VITA

- 1998 B.S. in Electronic and Computer Engineering,
Technical University of Crete, Greece
- 2000 M.S. in Computer Science,
University of California, San Diego
- 2005 Ph.D. in Computer Science,
University of California, San Diego

PUBLICATIONS

- M. Petropoulos, Y. Papakonstantinou, V. Vassalos: Graphical Query Interfaces for Semistructured Data: The QURSED System. In *ACM Transactions on Internet Technology (TOIT)*, 5(2), 2005.
- P. A. Bernstein, S. Melnik, M. Petropoulos, C. Quix: Industrial-Strength Schema Matching. In *ACM SIGMOD Record*, 33(4), 2004.
- M. Petropoulos, A. Deutsch, Y. Papakonstantinou: Query Set Specification Language (QSSL). In *Sixth International Workshop on the Web and Databases (WebDB)*, 2003.
- I. Zaslavsky, A. Memon, M. Petropoulos, C. Baru: Online Querying of Heterogeneous Distributed Spatial Data on a Grid. In *Third International Symposium on Digital Earth*, 2003.
- Y. Papakonstantinou, M. Petropoulos, V. Vassalos: Generating Query Forms and Reports for Semistructured Data: The QURSED Editor. In *Ninth Panhellenic Conference on Informatics (PCI)*, 2003.
- Y. Papakonstantinou, M. Petropoulos, V. Vassalos: QURSED: Querying and Reporting SEMistructured Data. In *ACM International Conference on Management of Data (SIGMOD)*, 2002.
- V. Hristidis, M. Petropoulos: Semantic Caching of XML Databases. In *Fifth International Workshop on the Web and Databases (WebDB)*, 2002.
- M. Petropoulos, Y. Papakonstantinou, V. Vassalos: Building XML Query Forms and Reports with XQForms. In *Computer Networks Journal, Special Issue on XML*, Elsevier Science, 39(5), 2002.
- M. Petropoulos, V. Vassalos, Y. Papakonstantinou: 10. XML Query Forms (XQForms): Declarative Specification of XML Query Interfaces. In *Tenth International World Wide Web Conference (WWW10)*, 2001.

ABSTRACT OF THE DISSERTATION

Exporting and Utilizing Database Interfaces on the Web

by

Michail Petropoulos

Doctor of Philosophy in Computer Science

University of California, San Diego, 2006

Database interfaces define the way database functionality is exported to and utilized by end users, developers and programs. Publishing, integration and service-oriented architectures demand capable interfaces and a higher degree of database functionality utilization in order to realize their potential.

In service-oriented architectures, applications need to provide integrated access to the data of multiple sources. Such applications typically support only a restricted set of queries over the schema they export, because the participating information sources contribute limited content and limited access methods. In prior work, these limited access methods have often been specified using a set of parameterized views, with the understanding that the integration system accepts only queries which have an equivalent rewriting using the views. These queries are called feasible. Infeasible queries are rejected without an explanatory feedback. To help a developer, who is building an integration application, avoid a frustrating trial-and-error cycle, I introduced the CLIDE query formulation interface, which extends the QBE-like query builder of Microsoft's SQL Server with a coloring scheme that guides the user toward formulating feasible queries. CLIDE

provides guarantees that the suggested query edit actions are complete (i.e. each feasible query can be built by following only suggestions), rapidly convergent (the suggestions are tuned to lead to the closest feasible completions of the query) and suitably summarized (at each interaction step, only a minimal number of actions are suggested).

In addition, applications need to publish powerful interfaces to end users who query and browse the underlying information sources. Such interfaces require extensive coding and are expensive to maintain. I developed the QURSED system, which semi-automatically generates web-based query form and report pages for semistructured XML data. QURSED drives the generation of the pages from the XML Schema describing the structure of the underlying data and offers to developers an authoring tool that does not require any coding. The resulting forms and reports encode large sets of parameterized queries and are powerful in the sense that heterogeneity, nesting and optionality are tackled declaratively. The system reduces the cost of developing and maintaining web applications by decoupling the query aspects from the visual ones.

Chapter I

Introduction

The context of this thesis is large-scale data integration and publishing systems. The architecture envisioned by database researchers for such systems is shown in Figure I.1. In the *Application Domain*, developers build applications that integrate several sources by making use of their exported content and access methods. Portals are the prominent examples at this level, such as `CNet.com` and `PCWorld.com`. In the *Source Domain*, shown at the bottom of the stack, sources export structure, content and access methods using web services. Typically, only a limited set of access methods are exported for business reasons, security constraints or technology limitations. Access methods are commonly expressed as (parameterized) database views of the sources' schema. For example, a *Dell* source provides `Computers` only if the desired `cpu` is given. Similarly, a *Cisco* source provides `Routers` if the `rate` is specified. The Source Domain has greatly benefited by the engineering advances in the popular area of service-oriented architectures [82]. For our purposes, we focus on data-oriented services embedded within standard WSDL web services [15].

The large number of sources and the large number of access methods inherently introduce heterogeneity in data formats, vocabularies and access methods. These phenomena prevents developers from having a unified view and access of the underlying data sources. The developer would like to acquire the information by

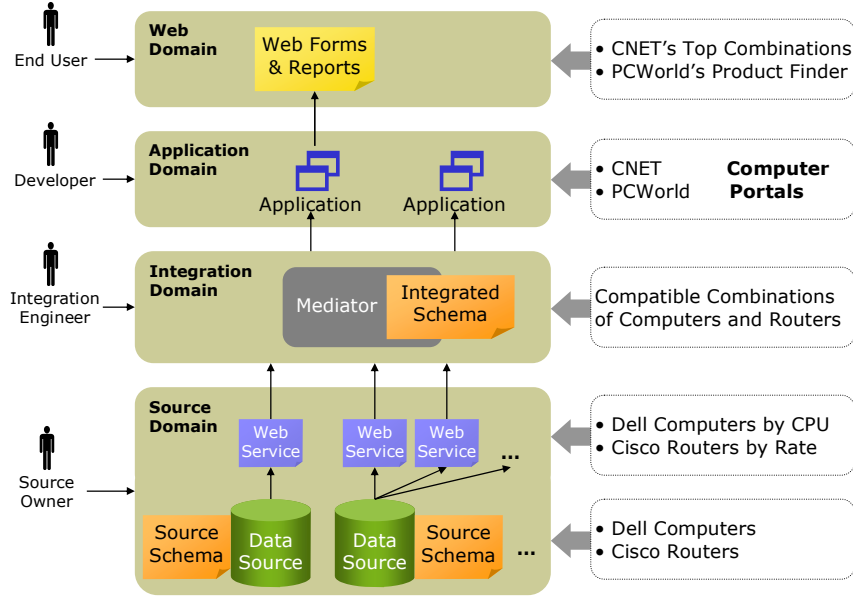


Figure I.1: Data Integration and Publishing Architecture

issuing a declarative query against an integrated schema. Not by specifying step-by-step with brute-force code or a workflow system how the data are to be obtained and put together. For that reason, a mediator [19, 26, 34, 52, 83] is employed in order to provide such a unified view.

In the *Integration Domain*, the integration engineer governs the process, inspects the services exported by the sources and constructs an integrated schema that satisfies the needs of the developers above. Through the integrated schema, the mediator provides a virtual view of the underlying sources to the developers. For example, an integrated schema describes compatible combinations of **Computers** and **Routers** exported by the Dell and Cisco sources. No data are stored, as opposed to the warehousing approach adopted through the use of ETL tools. Data are extracted from sources whenever a query is posed in an on-demand fashion. The mediator provides source transparency analogous to physical layer transparency in RDBMSs.

On the top-most layer, the developers need to publish information to users that do not use or understand query languages. In the *Web Domain*, query

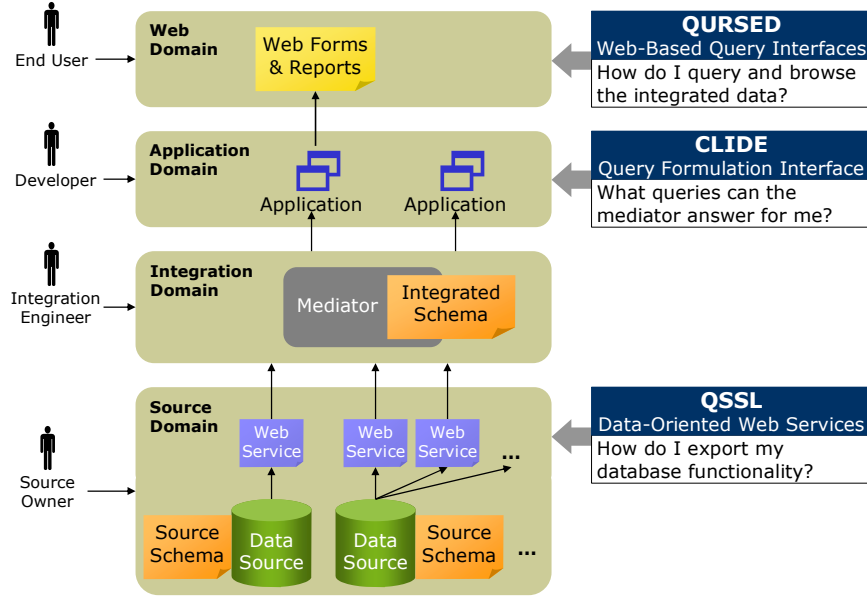


Figure I.2: Thesis Contributions

forms and reports are the typical interface between end-users and the underlying information sources. Browsing and querying arbitrarily complex data is the primary functionality offered by these interfaces, such as CNet’s *Top Combinations* page and PCWorld’s *Product Finder* page.

The information model of such architectures can be either relational, object-oriented, XML or a combination thereof.

Extensive work appears in the literature on the query languages, query processing, query reformulation, distributed query optimization and execution aspects of the integration and publishing problem [19, 26, 34, 52, 83, 46, 14]. This body of work did not prove sufficient to solve the integration problem in its entirety. The database community recognized several missing components that are needed in the process of implementing the architecture in Figure I.1 and are crucial to the adoption and applicability of existing works [61, 11]. Such components include descriptions of application interfaces and high-level model-driven tools that leverage these descriptions to help develop, integrate and evolve application systems. Moreover, the unification of web and database technologies are playing an

important role in the design and implementation of such components, since the web has become the default infrastructure for integration systems.

The results of this thesis drastically improve the applicability and usability of existing works in a wide range of scenarios by defining declarative user-oriented and application-oriented databases interfaces for the domains of the architecture in Figure I.1. More specifically, this thesis makes three contributions, shown in Figure I.2, in the Source, Application and Web Domains, respectively, which are described in detail below.

I.A Interactive Query Formulation Interfaces

Mediators provide to the Application Domain a single point of access over the participating sources by exporting a global schema against which application developers formulate their queries. When the participating sources export limited interfaces, such as web services, mediators are often capable to efficiently answer application queries by finding a rewriting that filters and combines the results of supported queries over the sources. An important issue that arises in such scenarios is how to expose the capabilities of the mediators. Since limited set of queries can be answered, application developers and users need to know what queries they can pose against a mediator, without being aware of the participating sources or examining their limited capabilities.

In prior work, the limited interfaces exported by the sources have often been specified using a set of parameterized views, with the understanding that the mediator accepts only queries which have an equivalent rewriting using the views. These queries are called feasible. Infeasible queries are rejected without an explanatory feedback. I introduced the CLIDE query formulation interface [63] to help a developer, who is building an integration application, avoid a frustrating trial-and-error cycle. CLIDE's architecture consists of a graphical front-end and a back-end. The front-end extends the query builder of Microsoft's SQL Server [81],

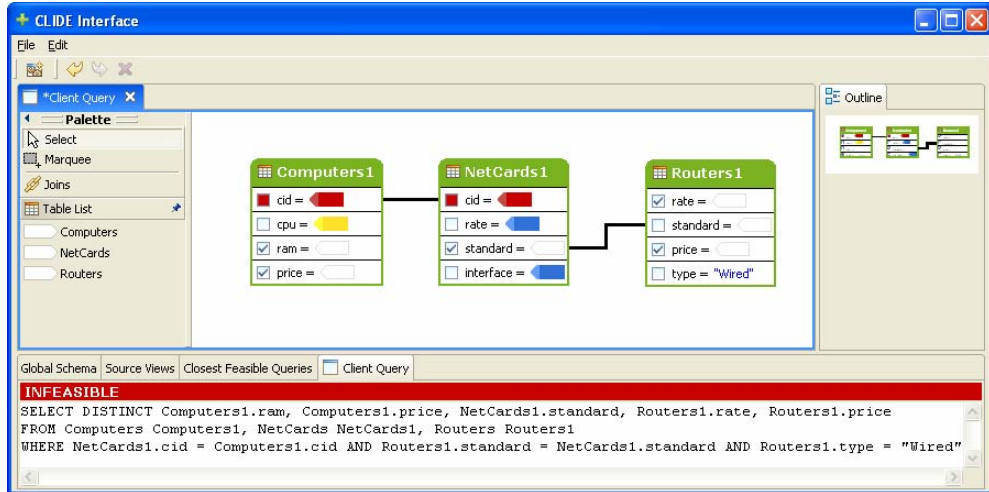


Figure I.3: CLIDE Front-End

which is based on the Query-By-Example (QBE) paradigm [95], with a coloring scheme that guides the user toward formulating feasible queries.

Figure I.3 displays CLIDE’s front-end and demonstrates the compactly-presented guidance, which in every step of the query formulation suggests which possible actions should, should not or may be taken in order to reach a feasible query. An action is the inclusion of a table in the **FROM** clause, the formulation of a selection condition in the **WHERE** clause or a projection of a column in the **SELECT** clause. The example of Figure I.3 is based on a global schema which is exported by a mediator and offers a unified view of the Dell and Cisco sources. The global schema consists of **Computers** and **NetCards**, coming from Dell, and **Routers**, coming from Cisco, and are shown on the left side of the CLIDE front-end.

The developer is formulating a query that returns **Computers**, **NetCards** and **Routers** with a compatible **standard**. Based on the limited interfaces exported by the Dell and Cisco sources, the red flag at the bottom indicates that the current query is infeasible and the colors indicate how to reach a feasible query, which will be a syntactic extension of the current one.

- *Yellow* color indicates actions that are necessary for the formulation of a feasible query. For example, conditioning the **cpu** of **Computers** is yellow

because all queries that the mediator can answer and involve the `Computers` table require a given `cpu`.

- *Blue* color indicates a set of actions where at least one of them is required to be taken in order to reach one of the next feasible queries. Notice that one can choose among many blue options. For example, one can perform a selection on the `rate` or the `interface` of `NetCards` in order to reach a feasible query.
- *Red* color indicates actions that lead to unsupported queries, regardless of what is included next. For example, conditioning the `cid` column of `Computers` or `NetCards` with a constant leads to unsupported queries.
- *White* color indicates selections, tables and projections whose participation in the query is optional.

Any good interface that guides the user toward some action must be comprehensive (complete) and, at the same time, avoid overloading the user with information at every step [59, 87]. CLIDE achieves both goals since it satisfies the following guarantees at every step of the interaction:

1. *Completeness of Suggestions*: Every feasible query can be built by following suggested actions only.
2. *Summarization of Suggestions*: The minimal set of actions that preserves completeness is suggested.
3. *Rapid convergence*: The shortest sequence of actions from a query to any feasible query consists of suggested actions.

Interaction sessions between the user and the CLIDE front-end are formalized using an *Interaction Graph*, which models the queries as nodes and the actions that the user performs as edges. Consequently, the color of each action is formally defined as a property of the set of paths that include the action and lead

to feasible queries. Then the above guarantees are formally expressed as graph properties.

The CLIDE back-end faces the challenge that the coloring properties cannot be trivially turned into an algorithm since they require the enumeration of an infinite number of feasible queries. Note that the number of queries is infinite for two reasons. First, there is an infinite number of constants that may be used. We tackle this problem by considering parameterized queries (similar to JDBC’s prepared statements) where each one stands for infinitely many queries. Still, the number of parameterized queries is infinite, because the size of the `FROM` clause of a SQL query is unlimited, which then leads to unlimited size `SELECT` and `WHERE` clauses.

I describe a set of algorithms that find a finite set of *closest feasible queries*, related to the current query, and determine the coloring by inspecting it. For my purpose, I leverage prior algorithms and implementations for finding exact and maximally-contained rewritings [69, 28, 73]. However, I needed to significantly optimize and extend current implementations in order to achieve on-line performance and to ensure that the produced maximally-contained queries are syntactic extensions of the current query, hence enabling the color algorithm. I provide a set of experiments that illustrate the class of queries and views CLIDE can handle, while maintaining on-line response.

In the current version, CLIDE targets relational databases and the SQL language, and the supported queries of the participating sources are specified by a set of parameterized conjunctive views with equality conditions, a formalism that has been widely used in integration scenarios. CLIDE’s modular architecture though is capable of accommodating many scenarios which would benefit from its approach to query building. One such scenario is data privacy enforcement. [74, 48] allow data owners to identify the non-sensitive data they are willing to export by means of parameterized, virtual views against the proprietary data. Data consumers formulate their queries against the proprietary database as well, but their

queries are rejected [74] or return null values [48] if they are not feasible according to the virtual views, which leads to a frustrating trial-and-error development process.

I.B Data-Oriented Web Service Interfaces

In the Source Domain, database owners need an interface definition language to export the limited access methods to their content. The popularity of service-oriented architectures has established the Web Services Description Language (WSDL) [15] as the default language for this purpose.

WSDL provides an XML format for describing functions published as web services. The function signatures typically have fixed numbers of input and output parameters. Although WSDL works well for arbitrary web services and in the general area of application integration, the “function” paradigm is not adequate when the software components behind the web services are databases. One typically associates one function with each parameterized query but this is problematic since databases often allow a large or even infinite set of parameterized queries over their schema. For example, the administrator of the Dell source may want to allow any query that selects **Computers** and **NetCards** by a combination of selection conditions on their attributes. Given that **Computers** and **NetCards** have 8 attributes combined, it is obviously impractical to specify 2^8 function signatures, even in our simple example.

In addition, the function paradigm does not state explicitly either the relationship between the input parameters and the output or the semantic connections the available functions have with each other and with the underlying database. We classify such *web services* as *functional* and we argue that they do not have sufficient expressive power to describe structurally rich and functionally powerful information sources, such as relational and emerging XML databases. For example, consider the following function signatures of two WSDL-based web

services exported by the Dell source, and the corresponding SQL queries that are executed when the services are called:

$CheapComputers(cpu) \rightarrow (Computer)^*$ (WS₁)

```
SELECT DISTINCT Com.cid, Com.cpu, Com.ram, Com.price      (Q1)
FROM Computers Com
WHERE Com.cpu=cpu
AND Com.price<='500'
```

$NetCards(rate) \rightarrow (NetCard)^*$ (WS₂)

```
SELECT DISTINCT Net.cid, Net.rate, Net.standard, Net.interface (Q2)
FROM NetCards Net
WHERE Net.rate=rate
```

Without the knowledge of the Dell schema, an application developer cannot be aware that the return types of these two services are part of the same database and that there is a foreign-key constraint on the `cid` of the two types, in which case it is meaningful to join them. Moreover, if the queries that are encapsulated within the function signatures are not published, then the developer is not aware that cheap computers according to Dell are the ones that are priced under \$500.

I proposed the *Query Set Specification Language (QSSL)* [64] for exporting web services on top of databases which overcomes the limitations of functional web services by exposing the schema of the underlying source (or view of the source) and a set of supported queries against this schema. QSSL is capable of concisely describing large sets of semantically meaningful parameterized queries, without requiring exhaustive enumeration of them. A QSSL specification is embedded in a WSDL specification, thus extending the current state-of-the-art instead of replacing it, to form a specialized type of web services, called *Data Services*.

QSSL targets the more demanding XML data model, in which case the schema of a data source is expressed in the XML Schema definition language [31]

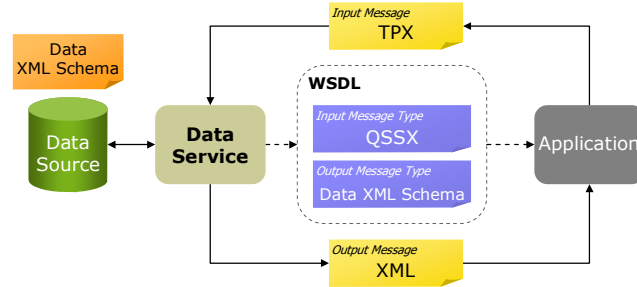


Figure I.4: Data Service Architecture

and the queries are tree pattern queries, which is a subset of XPath [53].

Figure I.4 shows the architecture of a Data Service published by a data source. The query capabilities exported by a Data Service provide an application with the means to formulate valid and acceptable queries and to be aware of the structure of the result. A Data Service receives an input message from the application and replies with an output message or a fault. The input message is a tree pattern (TP) query expressed in an XML format (TPX), and not just a set of parameters, and the output message is an XML tree. Hence the relationship between the input and output messages is explicit, since the input corresponds to a query and the output to its result. The set of acceptable tree pattern queries, i.e., the set of acceptable input messages, is a QSSL specification, which describes the possibly infinite set of parameterized queries that are acceptable. A QSSL specification is translated into an XML Schema (QSSX) describing the acceptable TPX messages.

I.C Web-based Query Form and Report Interfaces

In the Web Domain of Figure I.2, end-users need to query the integrated data, which are often complex and semistructured in nature. End-users are not expected to have knowledge of any specific query language, but are familiar with web-based forms and reports. For this purpose, I developed the QURSED system [62, 66, 67, 65] which generates and serves web-based query forms and re-

The screenshot shows a web browser window titled "Query Form and Report Pages - Microsoft Internet Explorer". The interface is divided into a left sidebar and a main content area.

Sensors - General

- Manufacturer: No preference (Balluff, Baumer, Turck)
- Sensing Distance: mm
- Protection Rating 1: NEMA3
- Protection Rating 2: No preference
- Operating Temperature: °C

Mechanical

- Body Type: No preference
- Dimension X: 20 mm
- Dimension Y: 40 mm

Report

- Results/page: 10
- Sort By Options: Body Type (ASC)
- Sort By Selections: DESC-Manufacturer, ASC-Sensing Distance
- Customize Presentation: Column Name, P, Image, Manufacturer, Part Number, Protection Ratings, Sensing Distance, Body Type (Diameter, Barrel Style, Height, Width)

Table Data (Next 10 items):

Image	Manufacturer	Part Number	Protection Ratings	Sensing Distance mm	Body Type
	TURCK	BC 3-M12-AN6X	NEMA1, NEMA3, NEMA4	6.0	cylindrical Diameter mm: 15, Barrel Style: Smooth
	TURCK	BC 3-M12-AP6X	NEMA3	6.0	cylindrical Diameter mm: 19, Barrel Style: Smooth
	TURCK	BC 5-Q08-AN6X2	NEMA3, NEMA4	7.0	rectangular Height mm: 14, Width mm: 9
	TURCK	BC 5-Q08-AP6X2	NEMA3, NEMA6, NEMA11	7.5	rectangular Height mm: 10, Width mm: 35
	TURCK	BC 5-S18-AN4X	NEMA3, NEMA11	10.0	rectangular Height mm: 15, Width mm: 10
	TURCK	BC 5-S18-AP4X	NEMA1, NEMA3, NEMA13	10.6	rectangular Height mm: 2, Width mm: 10
	TURCK	BC 5-S18-Y0X	NEMA1, NEMA3, NEMA11	12.0	rectangular Height mm: 17, Width mm: 23
	TURCK	BC 5-S185-AP4X	NEMA3	12.0	cylindrical Diameter mm: 10, Barrel Style: Threaded
	TURCK	BC10-M30-AZ3X	NEMA3, NEMA13	15.0	cylindrical Diameter mm: 11, Barrel Style: Smooth
	TURCK	BC10-M30-RZ3X	NEMA1, NEMA3, NEMA6	25.0	cylindrical Diameter mm: 20, Barrel Style: Threaded

Figure I.5: Example *QFR* Interface

ports for query processors that serve semistructured data and support the emerging XQuery language. The Enosys Application Builder [61] was a precursor of QURSED and part of a commercial integration platform.

Figure I.5 displays a QURSED-generated query form and report interface for proximity sensor products. End users interact with a query form page by typing or selecting constants in HTML form controls, such as text boxes and drop-down lists, and QURSED automatically formulates XQuery statements, which produce report pages. The form and report pages accommodate the intricacies of semi-structured (typically XML) data, i.e., data whose structure is characterized by high variability, nesting, repeatability and optional fields. For example, sensors are often

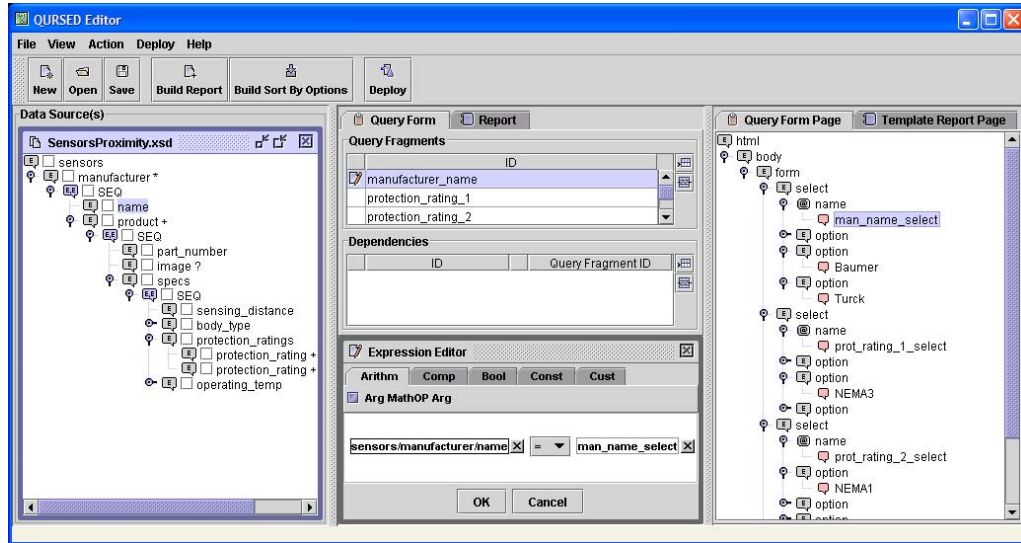


Figure I.6: The QURSED Editor

nested into manufacturer categories and each product of a sensor manufacturer comes with its own variations. Some sensors are rectangular and have height and width, and others are cylindrical and have diameter and barrel style. Some sensors have one or more protection ratings, while others have none.

QURSED is based on a Model Driven Architecture [1] and generates the pages from the XML Schema describing the structure of the underlying data. Moreover, an intuitive graphical interface, called the QURSED Editor, is provided to developers in order to semi-automatically produce such query forms and reports during design-time. Figure I.6 presents a snapshot of the QURSED Editor where the developer is presented with a visual representation of the XML Schema describing the underlying XML data on the left, and the query form page on the right. By performing visual actions, the developer associates schema elements with form controls on the query form page. The interface does not require programming or knowledge of XQuery. The end result is compiled and deployed to a QURSED-enabled application server.

The design of application-oriented interfaces that export semantically meaningful functionality poses many of the same challenges that user-oriented

interfaces pose. Hence, QURSED is based on a restricted version of QSSL, which declaratively specifies and describes a large set of parameterized queries that can be emitted from a query form page, and the complex structure of their results that is rendered on the report page. QSSL consists of a collection of fragments, each one capable of generating parts of an XQuery statement, typically navigations, selections and joins. The end user's interaction with the query form page implicitly activates the corresponding fragments, which at run-time are synthesized to automatically formulate the corresponding XQuery statement. QURSED renders on the report page the query result that is expressed directly in XHTML. Structural variance on the report page is tackled by producing heterogeneous rows/tuples in the resulting XHTML tables.

QURSED decouples the query aspects of the generated pages from the visual ones, hence making it easier to develop and maintain the resulting forms and reports whenever the underlying XML Schema changes - a labor-intensive and error-prone task with the current state of the art. Moreover, advanced capabilities are also offered, which provide fine-grained control on what queries make sense given the XML Schema and the semantics of the data.

I developed the techniques and algorithms employed by QURSED, with emphasis on how to accommodate the intricacies introduced by the semistructured nature of the underlying data. I specified the formal model of the query set specification, as well as its generation via the QURSED Editor, and focused on the techniques and heuristics the Editor employs for translating visual designer input into meaningful specifications. I also developed the algorithms QURSED employs for query generation and report generation.

I.D Thesis Overview

Chapter II presents CLIDE's front-end and formalizes its interaction with the developer using an interaction graph. Chapter III presents the architecture

of CLIDE's back-end and the corresponding algorithms for coloring actions on the front-end. Chapter IV covers the Query Set Specification Language (QSSL) and provides several examples where QSSL is applicable. Chapter V explains the architecture of QURSED and discusses the extensive related work. Chapter VI defines the restricted version of QSSL used by QURSED. Chapter VII elaborates on the visual steps the developer follows on the QURSED Editor to deliver query form and report interfaces. Chapter VIII concludes the thesis and discusses future work.

Chapter II

Interactive Query Formulation Interfaces

This chapter presents CLIDE’s front-end. Section II.A provides background information. Section II.B provides definitions and notation conventions. Section II.C discusses query building interfaces, focusing on CLIDE-related issues, and introduces the interaction graph, which allows us to formally define their behavior. Section II.D discusses the aspects of CLIDE that pertain to interaction in the presence of a limited set of feasible queries.

II.A Background

Many information sources support only a limited set of queries over their schema, as a result of privacy constraints [74, 48] or a result of limited access methods [94, 38]. In both privacy and mediation-oriented systems, a source specifies a set of queries that can be answered directly using views over its schema. A mediator extends the set of directly supported queries with a set of indirectly supported ones by appropriately rewriting the latter so that they are answered by filtering and combining the results of directly supported queries. If a submitted query is not supported the user simply receives a rejection, being forced into a trial-and-error query development loop. We propose that the user should be guided toward

feasible (i.e., supported) queries and we developed the CLIDE interactive system for this purpose.

The CLIDE (CLient guIDE) system is a graphical query formulation interface that mimics the visual paradigm of Microsoft’s Query Builder, incorporated in MS Access and MS SQL Server [81]. Microsoft’s Query Builder, in turn, is based on the Query-By-Example (QBE) [95] paradigm. CLIDE guides the user toward formulating feasible conjunctive queries and indicates any action that will lead toward a non-feasible conjunctive query. In particular, CLIDE provides compactly-presented guidance in the form of a color scheme, which in every step of the query formulation indicates which possible actions should, should not or may be taken in order to reach a feasible query. A flag indicates whether the current query is feasible or not. If it is, colors indicate how to reach another feasible query, which will be a syntactic extension of the current one. As usual, an action is the inclusion of a table in the **FROM** clause, the formulation of a selection condition in the **WHERE** clause or a projection of a column in the **SELECT** clause.

We illustrate the use of CLIDE and the color-driven interaction using an example from service-oriented architectures.

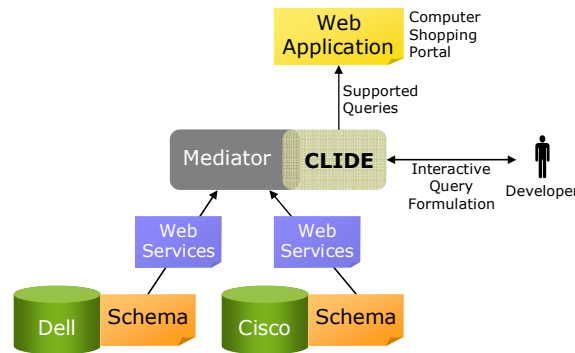


Figure II.1: Service-Oriented Architecture

Service-Oriented Architectures Information systems offer limited access to their data by publishing views, web services or APIs. For example, Amazon’s E-Commerce Service [2] provides a set of web services that allow one to query

its catalog and product data, and Google’s Web APIs [8] export web services for developers to issue search requests and receive results as structured data.

Service-oriented architectures [82] aggregate a collection of such services in order to provide more sophisticated web services and to support web applications. Figure II.1 shows a simple instance of an architecture where the mediator enables a computer shopping portal, such as CNET.com, to have integrated query access to two sources. We assume that Dell and Cisco export a set of web services on their computer and router catalogs, respectively. Since we want to be able to issue (distributed) queries, we associate schemas with Dell and Cisco and model the web services as parameterized views over those schemas [51]¹. Figure II.2 illustrates part of their respective schema and the signatures of four web services they export.

The Dell schema describes computers that are characterized by their cid, CPU model (e.g., P4), RAM installed and price, and have a set of network cards installed. Each network card has the cid of the computer it is installed in, accommodates a specific data rate (e.g., 54Mbps), implements a standards (e.g., IEEE 802.11g) and communicates with a computer via a particular interface (e.g., USB). The web service *ComByCpu* returns the computers of a given *cpu*. (We assume there is a *Computer* type.) The service *ComNetByCpuRate* provides computers of a given *cpu* that have installed network cards of a given data *rate*. The Cisco source describes routers that also accommodate a specific data rate, implement standards, have their own price and are of a particular type. The *RoutersWired* and *RoutersWireless* services return routers that are of either wired or wireless type respectively.

In Figure II.1, a user builds the computer shopping portal by formulating queries against the source schemas, and deploys a mediator in order to execute queries against the exported web services during run-time. The mediator can an-

¹Indeed, it is often the case that web services are based on parameterized queries over databases. However, for the purposes of mediation it is not necessary to assume that the Dell and Cisco schemas are known.


```

Computers(cid, cpu, ram, price)                                (Dell Schema)
NetCards(cid, rate, standard, interface)

ComByCpu(cpu) → (Computer)*
SELECT DISTINCT Com.*                                        (V1)
FROM Computers Com
WHERE Com.cpu=cpu

ComNetByCpuRate(cpu, rate) → (Computer, NetCard)*
SELECT DISTINCT Com.*, Net.*                                (V2)
FROM Computers Com, NetCards Net
WHERE Com.cid=Net.cid AND Com.cpu=cpu
AND Net.rate=rate

Routers(rate, standard, price, type)                          (Cisco Schema)

RoutersWired() → (Router)*
SELECT DISTINCT Rou.*                                       (V3)
FROM Routers Rou
WHERE Rou.type='Wired'

RoutersWireless() → (Routers)*
SELECT DISTINCT Rou.*                                       (V4)
FROM Routers Rou
WHERE Rou.type='Wireless'

(S1.Computers.cid, S1.NetCards.cid)                          (Column Associations)
(S1.NetCards.rate, S2.Routers.rate)
(S1.NetCards.standard, S2.Routers.standard)

```

Figure II.2: Source Schemas and Web Services

swer the query “return all P4 computers with a 54Mbps network card and the compatible wireless routers” by combining the answers of web service calls *ComNetByCpuRate* and *RoutersWired*. However, it cannot answer the query “return all computers with 1GB of RAM”. The reader is pointed to Chapter 20.3 of [38] for similar examples. CLIDE appropriately guides the user toward the formulation of feasible queries by employing the following coloring scheme:

- *Red* color indicates actions that lead to unsupported queries, regardless of what is included next. For example, conditioning the `type` column of `Routers` with a constant other than 'Wired' and 'Wireless' leads to unsupported queries.
- *Yellow* color indicates actions that are necessary for the formulation of a feasible query. For example, conditioning the `cpu` of `Computers` will be yellow since all queries that the mediator can answer and involve the `Computers` table require a given `cpu`.
- *Blue* color indicates a set of actions where at least one of them is required to be taken in order to reach one of the next feasible queries. Notice that one can choose among many blue options. For example, after the `cpu` of `Computers` has been conditioned and a feasible query has been reached, one should condition either the `ram` or the `price` column (among other choices) in order to reach the next feasible query.
- *White* color indicates selection conditions, tables and projections whose participation in the query is optional.

CLIDE is based on a modular architecture consisting of the front-end and a back-end that enables the front-end's behavior by deciding the color of each action. The above coloring scheme is implemented by CLIDE's front-end and is independent of the specification that has been used to describe the set of supported queries. Multiple back-ends are possible, depending on the nature of the specification of the supported query set. For example, one could have a back-end for the P3P privacy-related supported query set specification of [48] or the specification of [94] that is related to queries supported as a result of wrapping web forms.

Parameterized Views One of the most common and, at the same time, most challenging back-ends relates to the case where the set of directly supported queries are described using parameterized views, a technique that has been used to describe content and access methods in the widely used Global-as-View (GaV) integration

architectures [42], and also recently to describe privacy constraints in [74]. Going back to Figure II.2, the parameterized view V_1 corresponds to the web service *ComByCpu*. Notice that the parameterized view not only indicates the input (*cpu*) and output (*Computer*) of the service, but also indicates how the input and output are semantically related with respect to the underlying database. Typically the sources considered by the mediator can be too many to individually browse in order to formulate a feasible query.

Deciding whether a given query is feasible or not is a query rewriting problem: The mediator is given a query q over a database D and a set of parameterized views V_1, \dots, V_n and it searches for a plan (if any) that combines the views and computes $q(D)$. The plan is typically in the form of a query $q'(V_1(D), \dots, V_n(D))$ that runs on the views and often incorporates primitives that indicate the passing of information across sources and web services.

Several rewriting algorithms have been published; the reader is referred to the survey [42]. However, these algorithms are not sufficient for CLIDE’s backend since whenever there is no plan they only declare that the query cannot be answered. Some algorithms return overestimate or underestimate approximations of the query result, thus addressing a different goal than the one in our setting where the developer needs to know the exact queries that can be issued and program accordingly. Nevertheless, there are important technical connections between those algorithms and our work that are discussed in later sections.

II.A.1 Contributions

Formal Guarantees on the Interaction Any good interface that guides the user toward some action must be comprehensive (complete) and, at the same time, avoid overloading the user with information at every step [59, 87]. CLIDE achieves both goals since it satisfies the following guarantees at every step of the interaction:

1. *Completeness*: Every feasible query can be built by following suggested actions only.

2. *Minimality*: The minimal set of actions that preserves completeness is suggested.
3. *Rapid convergence*: The shortest sequence of actions from a query to any feasible query consists of suggested actions.

Interaction sessions between the user and the CLIDE front-end are formalized using an *Interaction Graph*, which models the queries as nodes and the actions that the user performs as edges. Consequently, the color of each action is formally defined as a property of the set of paths that include the action and lead to feasible queries. Then the above guarantees are formally expressed as graph properties.

Back-End Algorithms The challenge facing the CLIDE back-end is that the coloring properties cannot be trivially turned into an algorithm since they require the enumeration of an infinite number of feasible queries. Note that the number of queries is infinite for two reasons. First, there is an infinite number of constants that may be used. We tackle this problem by considering parameterized queries (similar to JDBC’s prepared statements) where each one stands for infinitely many queries. Still, the number of parameterized queries is infinite, because the size of the `FROM` clause is unlimited, which then leads to unlimited size `SELECT` and `WHERE` clauses.

We describe a set of algorithms that find a finite set of *closest feasible queries*, related to the current query, and determine the coloring by inspecting it. For our purpose, we leverage prior algorithms and implementations for finding exact and maximally-contained rewritings [69, 28, 73]. However, we needed to significantly optimize and extend current implementations in order to achieve on-line performance and to ensure that the produced maximally-contained queries are syntactic extensions of the current query, hence enabling the color algorithm. We provide a set of experiments that illustrate the class of queries and views CLIDE can handle, while maintaining on-line response.

CLIDE Demo We implemented the CLIDE front-end and the back-end algorithms which are available as an on-line demonstration at the anonymous address <http://www.clidedemo.com>.

II.B Definitions and Notations

The CLIDE front-end formulates queries from the set of conjunctive SQL queries with equality predicates $CQ^=$ under set semantics. The **FROM** clause consists of *table atoms* $R \ r$, where R is some table name and r an alias. The **SELECT** clause consists of the SQL keyword **DISTINCT** and *projection atoms* $r.x$, where x is a column of r . The **WHERE** clause is a conjunction of *selection atoms* and *join atoms*. *Constant* selection atoms are of the form $r.x=constant$, where r is some alias and x some column, while *parameterized* selection atoms are of the form $r.x=parameter$. Obviously, at most one selection atom for each alias-column pair can appear in the **WHERE** clause. Join atoms are of the form $r.x=s.y$. We define the *empty query* to have no table, join, selection or projection atom.

Column associations identify pairs of columns, within a source or across sources, whose join is meaningful. Figure II.2 illustrates the association of the **cid** columns of **Computers** and **NetCards** and the **rate** and **standard** columns of **NetCards** and **Routers**². The user can configure CLIDE to suggest either arbitrary joins or only joins between columns that are associated, in order to reduce the number of suggestions displayed to the user. In the latter case, the user still has the option to formulate joins between non-associated columns, but the CLIDE front-end will not suggest them. For the rest of the presentation, we assume the user has configured CLIDE to suggest joins between associated columns only. We denote this class of queries with $CQ^{=CA}$.

The views that CLIDE takes as input are from the set of parameterized conjunctive SQL queries $CQ^{=P}$, where *parameterized* selection atoms of the form

²Column associations can be explicitly declared by the mediator owner. They can also be derived from the pairs of type-compatible columns, from foreign-key constraints, the join atoms in the views, or any of the recently proposed schema matching techniques [72, 12].

$r.x=parameter$ appear in the WHERE clause. We assume that all joins are between associated columns. $CQ^{=CA}$ is a subset of $CQ^{=P}$.

Two queries q_1 and q_2 are *syntactically isomorphic*, denoted by $q_1 \cong q_2$, if they are identical modulo table alias renaming. Syntactic isomorphism is important since the users of query writing tools typically do not have control (or do not care to control) the exact table alias names.

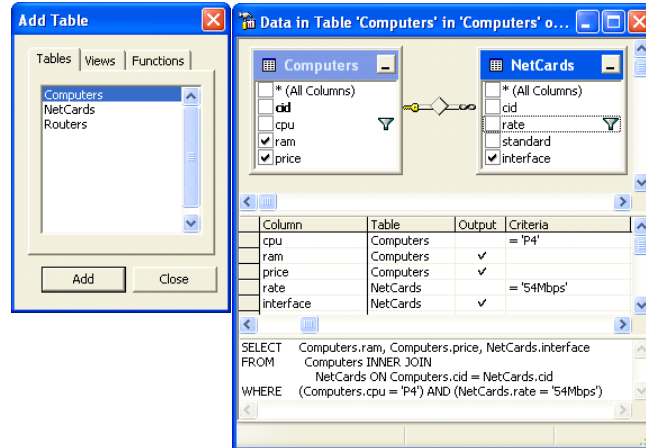
We denote the set of feasible queries by $FQ \subseteq CQ^{=CA}$. As in [73], we define the feasible queries given a set of views $\mathcal{V} = V_1, \dots, V_k \in CQ^{=P}$ over a fixed schema D , to be the set of queries $q_{F1}, \dots, q_{Fm} \in CQ^{=CA}$ over D that have an equivalent $CQ^{=}$ rewriting using \mathcal{V} . In the absence of parameters a rewriting is simply a query that refers exclusively to the views. In the presence of parameters we need to also ensure that there is a viable order of passing parameter bindings across the views of the rewriting [73, 75]. We capture this requirement as follows: First associate to each view a schema that includes both the columns that the view returns and the columns that participate in parameterized selections (even if they are not returned). Then we associate with each view schema a *binding pattern* that annotates every column that participates in a parameterized selections as *bound*, which is denoted by a ‘ b ’ superscript, and every other column as *free*, denoted by an ‘ f ’ superscript. For example, we associate the following schema and binding pattern to V_1 in Figure II.2:

$$V_1(\text{cid}^f, \text{cpu}^b, \text{ram}^f, \text{price}^f)$$

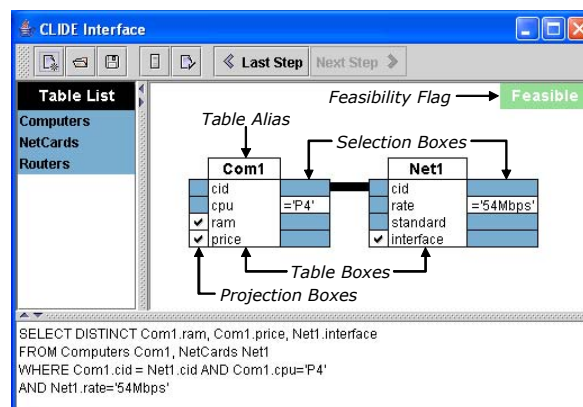
A *valid* rewriting is a query that refers to the views only and there is an order V_1, \dots, V_n of the used views such that if a column x is bound in V_i then either there is a selection atom $V_i.x=constant$ or a join atom $V_i.x=V_j.y$ where $j < i$.

II.C Query Building Interfaces

The CLIDE front-end is a QBE-like [95] graphical interface. It adopts Microsoft’s Query Builder interface [81] as the basis for the interactive query for-



(a) Microsoft's Query Builder



(b) CLIDE's Front-End expressing the above query

Figure II.3: QBE-Like Query Building Interfaces

mulation, since users are very familiar with it. Figure II.3a shows a snapshot of Microsoft's Query Builder, where the user formulates a query over the schemas of Figure II.2. The top pane displays the join of the `Computers` table with the `NetCards` table on `cid` and the projection of the `ram` and `price` columns of `Computers` and of the `interface` column of `NetCards`. The middle pane shows selections that set `cpu` equal to 'P4' and `rate` equal to '54Mbps', and the bottom pane displays the corresponding SQL expression. The user can add to the top pane tables from the list shown on the left. The user can also formulate joins, like the one on `cid`.

Figure II.3b provides a snapshot of CLIDE's front-end³ for the query of

³In this chapter we adopt a simplified version of the CLIDE's front-end in Figure I.3 for presentation purposes.

Figure II.3a. Apart from the feasibility flag and the coloring, the correspondence with Microsoft’s Query Builder is straightforward: CLIDE displays a *table box* for each table alias in the **FROM** clause. Selections on columns are displayed in *selection boxes*. Columns are projected using check boxes, called *projection boxes*. Joins are displayed as solid lines, called *join lines*, connecting the respective column names. The list of available tables is shown in a separate pane. Also shown is the SQL statement that the interface graphically represents. The “Last Step” and “Next Step” buttons allow the user to navigate into the history of queries formulated during the interaction.

The user builds $CQ^{=CA}$ queries with the following visual actions:

1. *Table action*: Drag a table name from the table list and drop it in the main pane. The interface draws a new table box with a fresh table alias and adds a table atom to the **FROM** clause of the SQL statement.
2. *Selection action*: Typing a constant in a selection box results to adding a selection atom to the **WHERE** clause.
3. *Join action*: Dragging a column name and dropping it on another one results to a join line connecting the two column names and a new join atom in the **WHERE** clause.
4. *Projection action*: Checking a projection box adds a projection atom to the **SELECT** clause.

II.D CLIDE Interaction in the Presence of Limited Access Methods

When not all $CQ^{=CA}$ queries against a database schema are feasible, CLIDE guides the user toward formulating feasible queries by coloring the possible next actions in a way that indicates what has to be done, what may and what cannot be done. Table actions are suggested by coloring the background of table

names in the table list. Selections and projections are suggested by coloring the background of their boxes. Joins are suggested by coloring join lines.

We illustrate the color scheme using the interaction session of Figure II.4, which refers to the running example of Figure II.2. The user wants to formulate a query that returns computers that meet various selection conditions, including conditions about network cards and routers - as long as those conditions are supported. Figure II.4 shows snapshots of the interaction session, where CLIDE's color scheme suggests, at each interaction step, which actions lead to a feasible query.

Required and Optional Actions Consider the query that the user has formulated in Snapshot 1. The interface indicates that this query is infeasible (see flag at top right) and that every feasible query that extends it must have a selection on `cpu`. The latter indication is given by coloring *yellow* (light gray on a B/W printout) the `cpu` selection box. The rest of the selection boxes and projection boxes are *white* suggesting that these actions are *optional*, i.e., feasible queries can be formulated with or without these actions being performed.

So the user performs the yellow selection on `cpu` by typing a constant in the selection box. This leads to the feasible query of Snapshot 2. This query is feasible since the mediator can run view V_1 with the parameter instantiated to 'P4' and then project out the `cid` and `cpu` columns.

Required Choice among multiple Actions The user may terminate the interaction session and incorporate the query of Snapshot 2 in her application or may continue to extend the query. The interface indicates that, in order to reach a next feasible query, at least one of the `NetCards`, `Routers` or (an additional) `Computers` tables has to be included in the query, among other options. The indication is provided by coloring the corresponding names in the table list *blue* (medium gray). Each given blue atom, say `NetCards`, does not appear in all feasible queries that extend the current query. If it did appear in all, then it would be yellow (i.e., required).

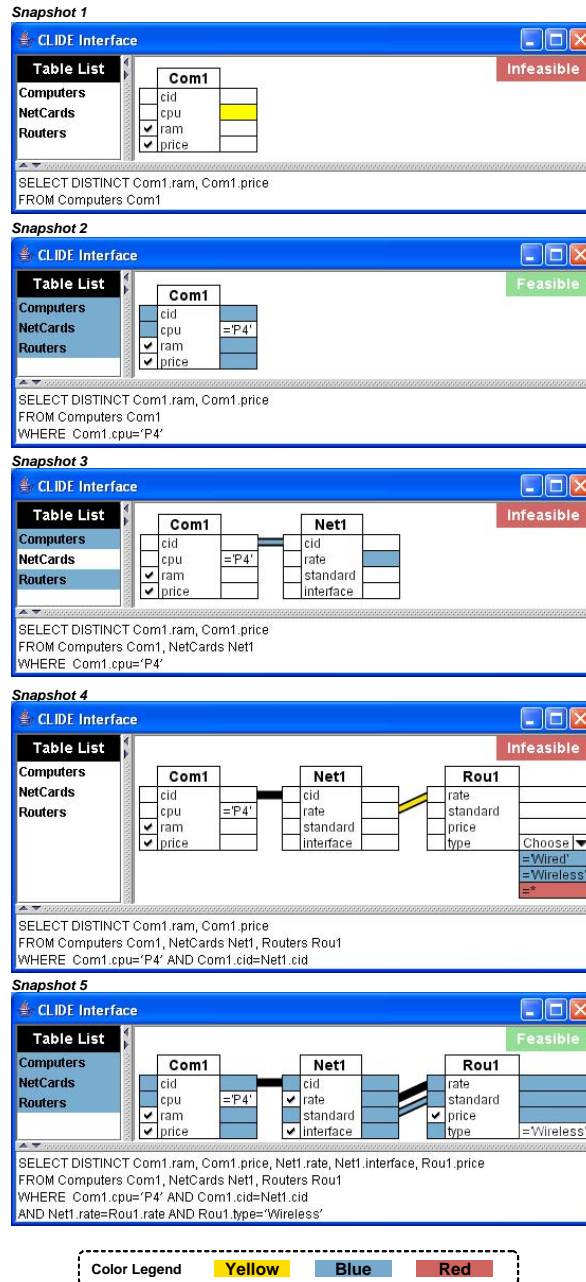


Figure II.4: Snapshots of an Interaction Session

Non-Obvious Feasible Queries Snapshot 3 presents a complex case, where the interface's color scheme informs the user about non-obvious feasible queries. After the user introduces a **NetCards** table, the interface suggests that one of the following extensions to the query is required: The join line between the *cid*'s of

`Computers` and `NetCards` is suggested since it leads to the formulation of view V_2 . It is blue since the user has more options: She can introduce a second copy of `Computers`, say `Com2`, which will lead toward the feasible query that joins `Networks` with `Com2`, selects on `rate` and takes a Cartesian product with `Com1`. If Cartesian product queries are of no interest to the user, she can set an option to have CLIDE ignore them. In such case the `cid` join would be a required (yellow) extension. For the remainder of the example, we assume that this option is set.

The user has another pair of options at Snapshot 3. She can perform the blue `rate` selection, which leads to the formulation of view V_2 . Alternatively, she may introduce a `Routers` table and join the `rate` columns of `NetCards` and `Routers`, thus instantiating the `rate` parameter of V_2 with constants provided by another table.

Selection Options In Snapshot 4, the user has performed the suggested join and introduced a `Routers` table. Now the `Routers.type` column needs to be bounded and the interface presents to the user a drop-down list that explains which constants may be chosen. She can either choose 'Wired' or 'Wireless'. The symbol * denotes any other constant and is colored *red* (dark gray) to indicate that no feasible query can be formulated if she chooses this option. Note that the options of a drop-down list can have different colors. If there were only one constant that she could choose, then this option would be yellow. In the special case where any constant can be chosen, then no drop-down list is shown, as in the case of the `cpu` selection box in Snapshot 1.

The front-end can also be configured to hide all red actions, including columns with red selection and projection boxes. Note that a red selection box implies a red projection box and vice versa. So the front-end can remove the column from the corresponding table box altogether.

In the next steps, the user performs the suggested join, chooses the 'Wireless' constant and checks several projection boxes. Snapshot 5 shows the new query, which is feasible. The mediator plan that implements this query first accesses view

V_4 , then for each `rate` returned accesses view V_2 with its parameters instantiated to ‘P4’ and the given `rate`, and finally performs the necessary projections.

The CLIDE front-end displays only yellow and blue join lines. Red and white join lines are typically too many and are not displayed. If the user wants to perform a join other than the ones suggested, she has to follow a trial-and-error procedure. Note that unchecked projection boxes can be either blue, white or red. A projection box cannot be yellow, because if there is a feasible query that has the corresponding projection atom in the `SELECT` clause, then the query formulated by removing this atom is also feasible.

Finally, if the user performs a red action, then all boxes, lines and items in the table list are colored red, indicating that the user has reached a dead end, i.e., no feasible query can be formulated by performing more actions and it is necessary to backtrack, i.e., undo actions.

II.D.1 Specification of CLIDE’s Color Scheme

Interaction sessions between the user and the CLIDE front-end are formalized by an *Interaction Graph*. The nodes of the interaction graph correspond to $CQ^{=CA}$ queries and the edges to actions.

Definition 1 *Interaction Graph* Given a database schema D and a set of $CQ^{=P}$ views \mathcal{V} over D , an interaction graph is a rooted DAG $G_I = (N, s, E)$ with labeled nodes and edges such that:

- For every query $q \in CQ^{=CA}$ over D there is exactly one node $n \in N$ whose label $q(n)$ is syntactically isomorphic to q . We call n feasible if $q(n)$ is feasible.
- s is the root node and is labeled with the empty query.
- E is the set of edges $e(n \xrightarrow{\mathbf{a}} n')$ labeled with an action \mathbf{a} which yields a query that is isomorphic to $q(n')$ when applied to $q(n)$. \mathbf{a} is either a table, a projection, a join, a specific selection of the form $\mathbf{r}.\mathbf{x}=\text{constant}$, or a generic selection

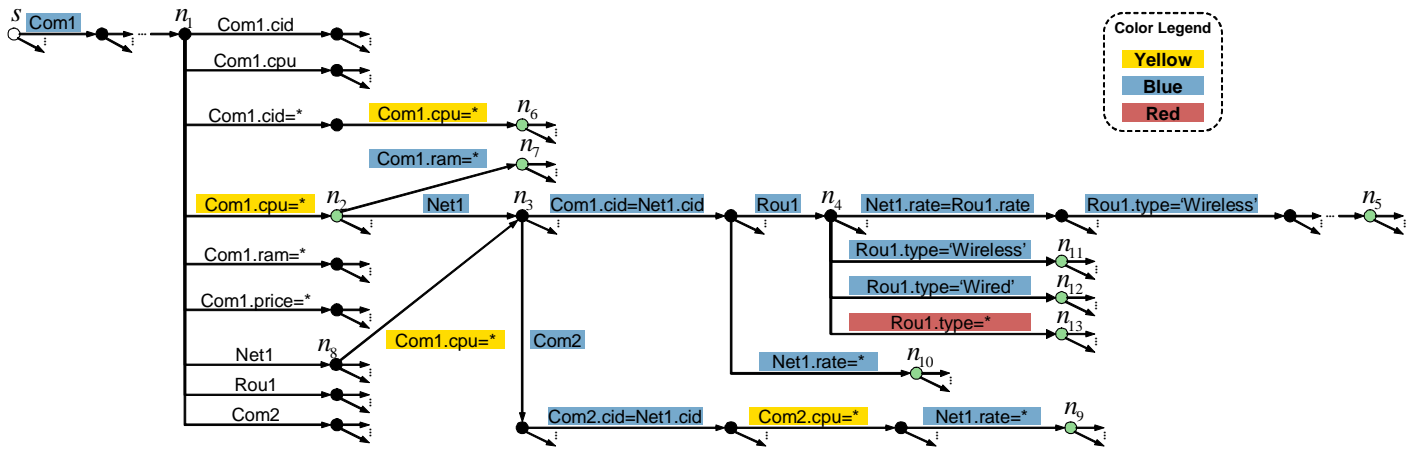


Figure II.5: Part of an Interaction Graph

of the form $r.x=*$. Here $*$ denotes any constant other than the ones that appear in specific selections and label edges originating from n .

Figure II.5 shows part of an interaction graph for the schemas in Figure II.2, where nodes n_1 to n_5 correspond to the queries formulated in Snapshots 1 to 5 of Figure II.4. Notice that there are multiple interaction graphs that correspond to a given schema, since each node n can be relabeled with any of the queries that are syntactically isomorphic to $q(n)$, i.e., with any query that uses other alias names. CLIDE considers a single interaction graph by controlling the generation of aliases. By convention, the generated aliases follow the lexical pattern Ti where T is the first three letters from the name of the table (for illustration purposes) and i is a number that is sequentially produced.

Figure II.5 indicates feasible queries by green (shaded) nodes. The root s is indicated by a hollow node. The outgoing edges of a node n capture all possible actions that the user can perform on $q(n)$. These are the actions that the front-end colors and they are finitely many. Even though there are infinitely many constants that can potentially generate infinitely many selections for a given column, they are aggregated by the $*$ symbol. In Figure II.5, for example, the $*$ in the selection `Com1.cpu=*` labeling an outgoing edge of n_1 aggregates all possible constants. The $*$ in the selection `Rou1.type=*` labeling an outgoing edge of n_4 denotes all constants except ‘Wired’ and ‘Wireless’, because corresponding selections label adjacent edges.

For a query $q(n)$, the coloring rules are formally expressed as a coloring of the actions labeling outgoing edges of node n .

Definition 2 *Colors* Given an interaction graph $G_I = (N, s, E)$, a node $n \in N$ and an outgoing edge $e(n \xrightarrow{a} m)$, the action a is colored:

- Yellow (*Required*) if every path p_i from n to a feasible node n_F contains an edge labeled with a .
- Blue (*At Least One Required*) if (i) a is not yellow, (ii) at least one path p_i

from n to a feasible node n_F contains an edge labeled with \mathbf{a} , and (iii) there is no path from n to n_F that contains a feasible node, excluding n and n_F .

- Red (*Forbidden*) if there is no path from n to a feasible node that contains an edge labeled with \mathbf{a} .
- White (*Optional*) if not colored otherwise.

We say that actions colored yellow or blue are called *suggested*. The same action may have different color at various points in the interaction. For example, table action `NetCards Net1` is white when it labels an outgoing edge of n_1 and blue when it labels an outgoing edge of n_2 .

CLIDE assigns colors according to Definition 2 and features the following characteristics of desirable behavior.

1. **Completeness of Suggestions** Every feasible query can be formulated by starting from the empty query and at every interaction step picking only among blue and yellow actions.
2. **Minimality of Suggestions** At every step, only a minimal number of actions, which are needed to preserve completeness, are suggested as required. Equivalently, for each blue or yellow action \mathbf{a} , there is at least one feasible query toward which no progress can be made without picking \mathbf{a} .
3. **Rapid Convergence by Following Suggestions** Assume that the user is at node n of the interaction graph and consequently follows a path p consisting of yellow and blue edges until she reaches feasible query $q(n')$. It is guaranteed that there is no path p' that is shorter than p and also leads from n to n' .

Chapter III

The CLIDE Back-End

This chapter presents the algorithms, implementation and performance evaluation of CLIDE’s back-end which shows that CLIDE is a viable on-line tool. Section III.A CLIDE’s back-end architecture. describes the algorithms of the CLIDE back-end. Section III.E describes the implementation and optimizations, which are experimentally evaluated in Section III.F. Section III.G presents related work and discusses CLIDE’s applicability to other settings. Section III.H provides proofs for the theorems and lemmas presented throughout the chapter.

III.A Architecture

The CLIDE back-end is invoked every time the interaction arrives at a node n in the interaction graph. It takes as input the query $q(n)$, the schemas and the views exported by the sources, and the set of column associations. The back-end partitions the set of possible actions, which label outgoing edges of n , into sets of blue, red, white and yellow suggested actions. It also decides if $q(n)$ is feasible or not.

The first challenge in determining the partition is that the color definitions make statements about all possible extensions of the current query, i.e., all feasible nodes reachable from n . These correspond to an infinite set of infinitely long paths in the interaction graph. Hence, the color definitions cannot be trivially translated

into an algorithm.

We show that at each interaction step, it is sufficient to consider only a representative subgraph of the interaction graph to color the possible actions either blue or yellow. This subgraph consists of n , the feasible nodes that are closest to n , and the paths connecting n to these feasible nodes. The closest feasible nodes are labeled with queries in $FQ_C(n)$ which is defined below.

Definition 3 *Closest Feasible Queries FQ_C* Given an interaction graph $G_I = (N, s, E)$ and a node $n \in N$, the set of closest feasible queries $FQ_C(n)$ are the ones that label feasible nodes n_F reachable from n such that there is no path p from n to n_F that contains a feasible node, excluding the endpoints of p .

Section III.B presents the computation of $FQ_C(n)$ when parameterized selection atoms do not appear in the views. We show that $FQ_C(n)$ is finite and present optimizations for computing it, which proved crucial to CLIDE’s usability. If parameterized selection atoms appear in the views, then $FQ_C(n)$ is infinite. Section III.D shows that CLIDE’s back-end faces this additional challenge without compromising any of the formal guarantees by computing a finite representative set of *seed queries* $SQ(n)$.

The second challenge (Section III.C) that the back-end faces is to efficiently color the possible actions given the set of closest feasible queries. Even though coloring an action yellow or blue is straightforward and inexpensive, coloring the remaining actions red or white using a brute force algorithm leads to significant performance overhead.

Figure III.1 shows the architecture of the CLIDE system implementation. Currently, the system parses the schemas, view definitions and column associations from corresponding text files. The *Closest Feasible Queries Algorithm* computes the set $FQ_C(n)$ and implements the algorithm of Section III.B. When parameterized selection atoms do not appear in the views, the *Color Algorithm* component inputs the set $FQ_C(n)$ and implements the algorithm of Section III.C. When

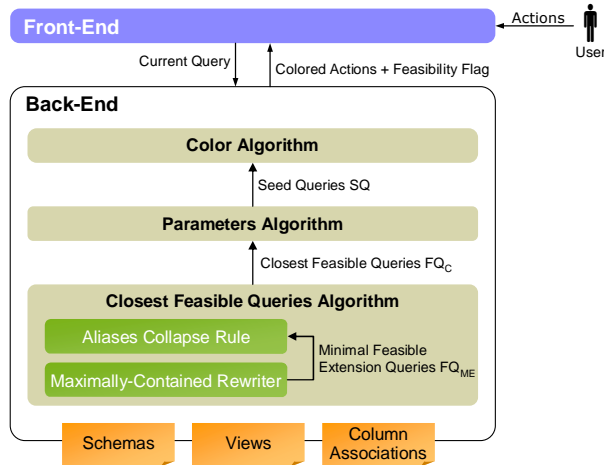


Figure III.1: CLIDE Architecture

parameterized selection atoms appear in the views, the *Color Algorithm* component inputs a set of *seed queries* $SQ(n)$ produced by the *Parameters Algorithm* component described in Section III.D.

III.B Closest Feasible Queries Algorithm

The search for closest feasible queries faces an infinite search space, namely all possible extensions of the current query. We limit this space to a finite one, corresponding to nodes in the interaction graph that are within a bounded distance from n . Then, we present an efficient method for enumerating $FQ_C(n)$ without exploring the whole search space.

Maximally-Contained Feasible Queries Intuitively, as the user syntactically extends the current query with new tables, selections and joins, she creates queries which are contained in the initial one. It is therefore a natural starting point to search for the closest feasible queries among the contained and feasible ones. We can further focus on the maximally-contained [42] and feasible queries since they are the least constraining (semantically) and hence they have the least number of additional tables, selections and joins. As in [42], the set of maximally-contained feasible queries is formally defined as the set of queries such that

1. for each maximally-contained query q_1 , q_1 is feasible and contained in $q(n)$ ($q_1 \sqsubseteq q(n)$),
2. for each maximally-contained query q_1 and any feasible query q'_1 , if q'_1 contains q_1 , then q'_1 is equivalent to q_1 , and
3. for each feasible query $q_1 \sqsubseteq q(n)$ there exists a maximally-contained query q_2 such that $q_1 \sqsubseteq q_2$.

Among the maximally-contained feasible queries, we focus on the ones which are *minimal syntactic extensions* of $q(n)$, in the sense that dropping any table, selection or join compromises feasibility or containment in $q(n)$ or the property of syntactically extending $q(n)$. We denote this set as $FQ_{ME}(n)$. Section III.E describes how we extended one of the several maximally-contained rewriting algorithms proposed in the literature [42] to obtain $FQ_{ME}(n)$.

$FQ_{ME}(n)$ is known to be finite if we restrict $q(n)$ and the views to conjunctive queries with constant selection atoms [42].

Lemma III.B.1 *All minimal feasible extensions of $q(n)$ which are maximally-contained are also closest feasible queries ($FQ_{ME}(n) \subseteq FQ_C(n)$).* \diamond

However, there are closest feasible queries that are not in $FQ_{ME}(n)$, as the next example shows, and we will have to find them.

Example 4 *Assume that views V_1 and V_2 of Figure II.2 are replaced by the following views V'_1 and V'_2 , respectively, which contain constant selections only.*

```
SELECT DISTINCT Com.*
FROM Computers Com
WHERE Com.cpu='P4'
```

(V'₁)

```
SELECT DISTINCT Com.*, Net.*
FROM Computers Com, Network Net
WHERE Com.cid=Net.cid AND Com.cpu='P4'
AND Net.rate='54Mbps'
```

(V'₂)

If the current query is $q(n_3)$ in Figure II.5 (Snapshot 3 in Figure II.4), then the only query in $FQ_{ME}(n_3)$ is $q(n_9)$ given below.

```

SELECT DISTINCT Com1.ram, Com1.price                                q(n9)
FROM Computers Com1, Computers Com2, NetCards Net1
WHERE Com2.cid = Net1.cid AND Com1.cpu='P4'
AND Com2.cpu='P4' AND Net1.rate='54Mbps'

```

Note that $q(n_{10})$ is also a closest feasible query to $q(n_3)$, but it is not in $FQ_{ME}(n_3)$ since it is contained in $q(n_9)$.

```

SELECT DISTINCT Com1.ram, Com1.price                                q(n10)
FROM Computers Com1, NetCards Net1
WHERE Com1.cid = Net1.cid AND Com1.cpu='P4'
AND Net1.rate='54Mbps'

```

*Intuitively, one can extend $q(n_9)$ with joins until the **Com2** alias “collapses” into **Com1**, leading to a closer query, reachable from $q(n_3)$ and clearly contained in $q(n_9)$ due to the added joins.*

Even though $FQ_{ME}(n)$ does not give us the set of closest feasible queries, we can use it to bound the search space for $FQ_C(n)$. Theorem 5 below states that all queries in $FQ_C(n)$ correspond to nodes located within a bounded distance from n .

Theorem 5 *Given a node n in the interaction graph and the set $FQ_{ME}(n)$, if p_L is the longest path from n to a node labeled with a query in $FQ_{ME}(n)$, then all nodes labeled with queries in $FQ_C(n)$ are reachable from n via a path p , where $|p| \leq |p_L|$.*

Theorem 5 enables a brute force algorithm for computing $FQ_C(n)$: (i) compute $FQ_{ME}(n)$, (ii) compute the bounded distance $|p_L|$ as the length of the longest path from n to some query in $FQ_{ME}(n)$, (iii) enumerate the set of queries $B(n, |p_L|)$ reachable from $q(n)$ by systematically applying up to $|p_L|$ actions in all possible ways, and (iv) return all feasible queries from $B(n, |p_L|)$.

This algorithm computes $FQ_C(n)$, but is highly inefficient. In the worst case, it enumerates all paths of length $|p_L|$. The following observations allow us to prune this search dramatically, by starting from $FQ_{ME}(n)$.

Alias Collapse Rule We can compute $FQ_C(n) \setminus FQ_{ME}(n)$ starting from the queries in $FQ_{ME}(n)$ and rewriting them using the *alias collapse rule*, which rewrites a query q into a query q' as follows: pick a pair of table atoms sharing the same relation name, say R R_1 , R R_2 , and rename R_2 with R_1 in q , to obtain q' .

Example 6 *One can obtain the closest feasible query $q(n_{10})$ from query $q(n_9)$ by collapsing the aliases Com1 and Com2.*

Notice that indiscriminate application of the collapse rule can lead to unsatisfiable queries. To see this, assume that q contains the selection conditions $R_1.x='5'$ and $R_2.x='3'$. After collapsing R_1 and R_2 , q' contains the inconsistent selection conditions $R_1.x='5'$ and $R_1.x='3'$. We apply the alias collapse rule only if they lead to satisfiable queries.

Lemma III.B.2 *For any $q_1 \in FQ_C(n) \setminus FQ_{ME}(n)$, there exists $q_2 \in FQ_{ME}(n)$ such that q_1 is obtained from q_2 by repeatedly applying the alias collapse rule. \diamond*

Lemmas III.B.1 and III.B.2 lead to the following algorithm for computing $FQ_C(n)$.

algorithm Quick FQ_C

Input: node n

Output: $FQ_C(n)$

begin

compute $M := FQ_{ME}(n)$ using an algorithm for finding maximally-contained rewritings,
extended to produce minimal syntactic extensions of $q(n)$

// compute $FQ_C(n) \setminus FQ_{ME}(n)$ in AC :

let $AC :=$ the empty set \emptyset

for each $q_M \in M$ do

 for each pair of distinct aliases r_1, r_2 of some relation in q_M do

 let $q :=$ collapse r_1 and r_2 in q_M

$AC :=$ **CollapseToFeasible**(q, AC)

return $M \cup AC$

end

procedure CollapseToFeasible

Input: query q , query set AC
Output: all feasible queries obtainable from q by collapsing aliases
begin
 if q is unsatisfiable return the empty set \emptyset

 if q is feasible and not contained in any $q_i \in AC$
 $AC := AC \cup \{q\}$
 for each pair of distinct aliases r_1, r_2 of some relation in q do
 let $q' := \text{collapse } r_1 \text{ and } r_2 \text{ in } q$
 $AC := \text{CollapseToFeasible}(q', AC)$
 return AC
end

Theorem 7 $\text{Quick}FQ_C$ computes $FQ_C(n)$.

III.C Color Algorithm

After computing the set of closest feasible queries $FQ_C(n)$, CLIDE decides if the current query is feasible or not, and then colors all possible actions that the user can perform next. The current query is feasible if it is a closest feasible one, i.e., $q(n) \in FQ_C(n)$, and infeasible otherwise. We first present the algorithm for finding the yellow and blue actions when the current query is infeasible. We deal with the white and red actions, as well as the feasible case, next.

Blue and Yellow Instead of working with the infinite interaction graph, we can restrict our attention to the finite *close subgraph* consisting of n , all closest feasible nodes labeled with the closest feasible queries in $FQ_C(n)$ and the paths between them. Then we have:

Lemma III.C.1 *For an infeasible current query $q(n)$, and for every action \mathbf{a} applicable to $q(n)$, \mathbf{a} is colored yellow (blue) with respect to the interaction graph if and only if \mathbf{a} is colored yellow (blue) with respect to the close subgraph of n . \diamond*

At this point, it is easy and more efficient to color the actions without actually materializing the close subgraph. We color a join \mathbf{a} yellow if it appears in all closest feasible queries, and blue if it appears in some. In the case of a table

action T , we color it yellow (resp. blue) if in all (resp. some) closest feasible queries there exists a table atom T_j , such that T_i and T_j do not necessarily refer to the same alias, and T_j does not appear in the current query.

Specific selections, i.e., selections of the form $r.x=constant$, are colored either yellow or blue the same way joins are colored. The front-end displays these actions in the corresponding selection box as options of a drop-down list. *Generic selections* of the form $r.x=*$ and projections cannot be colored blue or yellow when the current query is infeasible, because for each feasible query they participate in, there is another feasible query that can be formulated without performing them. Conversely, when the current query is infeasible, performing a projection or a generic selection that does not appear in the views will not yield a feasible query¹.

White and Red Any remaining actions are either white or red. For each such action a , a brute force approach would add a to the current query, thus yielding query $q(n')$, and then test if $FQ_C(n')$ is empty. If so, a is colored red, otherwise white. This approach, although simple, requires the non-emptiness test of $FQ_C(n')$, which is an expensive operation, as the experiments of Section III.F demonstrate. Hence, we need to devise more efficient techniques for coloring red and white actions.

In the case of table actions we color red the ones that are not used in any view, and white the remaining ones, since a feasible query q_F can lead to another feasible query that takes the Cartesian product of q_F and the view that contains the table in question.

For the case of projections and selections, we attach a *maximum projection list* to every closest feasible query $q_F \in FQ_C(n)$. A maximum projection list consists of all projections that can be added to q_F , in addition to the ones already in the current query, without compromising feasibility. For example, if we add all possible projections to $q(n_9)$ of Example 4, while preserving feasibility, then we formulate the following query $q'(n_9)$:

¹Note that generic selections can be colored yellow or blue when parameterized selections appear in the views. Please see Section III.D for details.

```

SELECT DISTINCT
    Com1.cid, Com1.cpu, Com1.ram, Com1.price
    Com2.cid, Com2.cpu, Com2.ram, Com2.price
    Net1.cid, Net1.rate, Net1.standard, Net1.interface
FROM Computers Com1, Computers Com2, NetCards Net1
WHERE Com2.cid = Net1.cid AND Com1.cpu='P4'
AND Com2.cpu='P4' AND Net1.rate='54Mbps'

```

$q'(n_9)$

Hence, the maximum projection list of $q(n_9)$ consists of all projections in $q'(n_9)$ except `Com1.ram` and `Com1.price` which appear in $q(n_9)$. In Section III.E we show how we extended a maximally-contained rewriting algorithm to generate these lists in linear time.

Once we compute the maximum projection lists, we color a projection red if it does not appear in any list. Generic selections are colored red if the projection `r.x` is red. These selections are also shown as options of the corresponding drop-down lists. In the special case where no specific selections exist, then no drop-down list is displayed and the selection box is colored according to the color of the generic selection.

Any remaining actions are colored white. Note that specific selections can never be colored white or red. The CLIDE front-end does not display white and red joins, so they are not a consideration.

Feasible Current Query If the current query is feasible, we still use the same algorithm, but we color all non-red actions blue, as each one leads to a new feasible query, not obtainable via other actions.

III.D Parameters

When parameterized selection atoms appear in the views, the algorithms in Sections III.B and III.C need to be extended, because the set of closest feasible queries becomes infinite. The following example illustrates this point.

Example 8 *Assume the following employees and managers source schema. The exported parameterized view V_5 returns the `mid` of an employee's manager, given*

the employee's `eid`. V_6 returns the `salary` of a manager, given the manager's `mid`. Note that the source schema is recursive, i.e., an employee has a manager, but a manager is also an employee, who has a manager. One of the column associations we consider witnesses this recursion.

```
Empls(eid, mid)                                     (Schema)
Mngrs(mid, salary)
```

```
EmplsMngrs(eid) → (Employee)*
SELECT DISTINCT E1.*                               (V5)
FROM Empls E1
WHERE E1.eid=eid
```

```
MngrsSalary(mid) → (Manager)*
SELECT DISTINCT M1.*                               (V6)
FROM Mngrs M1
WHERE M1.Mid=mid
```

```
(S1.Empls.eid, S1.Empls.mid)                       (Column Associations)
(S1.Empls.mid, S1.Mngrs.mid)
```

The user wants to find out the salaries of an employee's managers and has currently formulated query q_1 :

```
SELECT DISTINCT M1.salary                           q1
FROM Mnrgs M1, Empls E1
WHERE M1.mid=E1.mid
```

At this point, `E1.eid` has to be provided to reach a feasible query. Therefore, the front-end makes two suggestions: (i) perform a selection on `E1.eid`, or (ii) introduce a second `Empls E2` table, so that parameters can be passed from `E2.mid` to `E1.eid` (based on the first column association). The suggested actions are both blue.

Option (i) will formulate the feasible query q_{2F} which returns the salaries of `E1.eid` employee's immediate managers.

```
SELECT DISTINCT M1.salary                           q2F
FROM Mnrgs M1, Empls E1
WHERE M1.mid=E1.mid
AND E1.eid='A123'
```

Option (ii) leads toward a query that returns the salaries of managers that are two levels above an employee. More specifically, if the user introduces table a second table **Empls E2**, then the front-end colors the join **E1.eid=E2.mid** yellow, which formulates q_3 :

```
SELECT DISTINCT M1.salary q3
FROM Mnrgrs M1, Empls E1, Empls E2
WHERE M1.mid=E1.mid AND E1.eid=E2.mid
```

For q_3 , the front-end makes the same kind of suggestions to the user as for q_1 , since **E2.eid** has to be provided now. A selection on **E2.eid** formulates the feasible query q_{4F} which returns the salaries of managers that are two levels above that employee.

```
SELECT DISTINCT M1.salary q4F
FROM Mnrgrs M1, Empls E1, Empls E2
WHERE M1.mid=E1.mid AND E1.eid=E2.mid
AND E2.eid='A123'
```

It becomes evident that the user can build chains of **Empls** aliases of an unbounded length, where each alias joins its **eid** with the next one's **mid**, before performing a constant selection on the **eid** of the last **Empls** alias. These queries are infinitely many and are all closest feasible. For example, q_{2F} and q_{4F} are two such queries, and there is no sequence of actions that applied on q_{2F} formulate q_{4F} .

Searching for the closest feasible queries starting from the maximally-contained ones becomes problematic as it is known that the latter set is infinite in the presence of binding patterns [42]. Moreover, the coloring of actions cannot be done by enumerating all closest feasible queries.

Instead, CLIDE identifies a finite set of parameterized *seed queries* $SQ(n)$, where $q(n)$ is the current query. These are not necessarily feasible, but have the property that each path toward a closest feasible query must pass through some seed query first. In Example 8, q_{2F} is a feasible seed query of q_1 , while q_3 is an infeasible one, which however must be constructed on the way to q_{4F} . The algorithm

suggests to the user a finite set of actions leading from $q(n)$ toward the seed queries $SQ(n)$. This can be done by simply calling the color algorithm of Section III.C on $SQ(n)$ instead of $FQ_C(n)$. This approach does not compromise the guarantees of completeness, minimality of suggestions and rapid convergence.

It is a priori non-obvious that the finite set $SQ(n)$ even exists. However, it turns out that this is indeed the case, and moreover that $SQ(n)$ can be computed as follows. Start by ignoring the binding patterns of the views and computing the maximally-contained rewritings of $q(n)$ in terms of the views. Under the original binding patterns, not all obtained rewritings are valid, and the values of their parameters must be provided. In each such rewriting, parameter values may be provided by (i) selections with a constant, or (ii) via a parameter-passing join with a view alias from within the rewriting or (iii) via a parameter-passing join with a new view alias. The considered parameter-passing joins must be compatible with the column associations. Notice that there are only finitely many considered selections and parameter-passing joins. We obtain $SQ(n)$ by systematically extending the rewritings according to possibilities (i), (ii) and (iii), and unfolding the view definitions in all extended rewritings.

III.E Implementation

The current implementation of CLIDE consists of the components that compute the closest feasible queries and color the actions, shown in Figure III.1. The component that handles parameterized selections in the views is under development.

CLIDE uses MiniCon [69] as the core of its maximally-contained rewriting component. Even though an initial implementation was provided to us, we had to significantly optimize and extend it in order to enable CLIDE’s color algorithm and achieve on-line performance. Figure III.2 illustrates the anatomy of the maximally-contained rewriting component from Figure III.1.

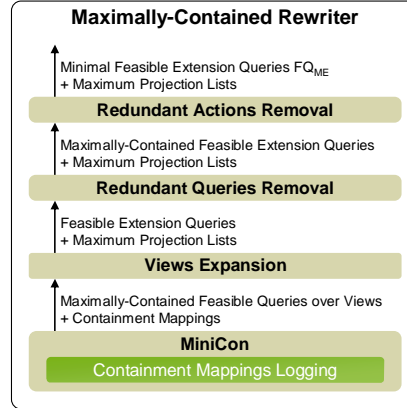


Figure III.2: MiniCon Optimizations and Extensions

Views Expansion The first challenge we faced was that MiniCon does not produce maximally-contained rewritings that are syntactic extensions of the current one. MiniCon initially produces a set of rewritings expressed using the views. Once these rewritings are expanded so that they are expressed in terms of the source schemas, they are not syntactic extensions of the current query, because fresh aliases are introduced. For example, if the current query is $q(n_3)$ (Snapshot 3 in Figure II.4), MiniCon produces the following rewriting query q_R that combines V'_1 and V'_2 :

```
SELECT DISTINCT V'_1.ram, V'_1.price                                q_R
FROM V'_1, V'_2
```

After expanding the views of q_R , we obtain the following query q_{RE} , which is expressed in terms of the source schemas.

```
SELECT DISTINCT ComA.ram, ComA.price                                q_RE
FROM Computers ComA, Computers ComB, NetCards Net
WHERE ComB.cid = Net.cid AND ComA.cpu='P4'
AND ComB.cpu='P4' AND Net.rate='54Mbps'
```

Query q_{RE} is syntactically isomorphic to the closest feasible query $q(n_9)$, but it is not a syntactic extension of $q(n_3)$, since $q(n_3)$ contains a table `Computers Com1`, while q_{RE} contains the tables `Computers ComA` and `Computers ComB`. It is not straightforward if `Com1` corresponds to `ComA` or `ComB`.

We could find the correspondences between the tables of $q(n_3)$ and the tables of q_{RE} by computing the containment mapping [4] from $q(n_3)$ into q_{RE} . The containment mapping considers all atoms of the two queries in order to find the correct correspondences. For example, `Com1` of $q(n_3)$ cannot be mapped into `ComB` of q_{RE} , because the `ComB.ram` and `ComB.price` projections do not appear in the `SELECT` clause, as is the case in $q(n_3)$. Once we compute the containment mappings, we can turn the MiniCon rewriting queries into syntactic extensions of the current query by renaming the aliases of the former.

We managed to avoid computing the containment mappings on top of MiniCon. We observed that while MiniCon searches for maximally-contained rewritings, it builds the containment mappings from the current query to the maximally-contained ones. So we extended MiniCon to log this information and output it along with the set of maximally-contained rewriting queries over the views, as shown at the bottom of Figure III.2.

Subsequently, we wrote a *Views Expansion* component, which uses the logged containment mappings to expand the views in every MiniCon maximally-contained rewriting so that the resulting queries are syntactic extensions of the current one.

The *Views Expansion* component also generates the maximum projection lists used in the color algorithm of Section III.C. In Section III.C, we defined a maximum projection list to be the list of all possible projections that can be added to a query without compromising feasibility. It turns out that for each expanded query, the maximum projection list corresponds to all projections in the views that appear in the initial MiniCon rewriting. For example, the initial rewriting of $q(n_9)$ is q_R . We can safely add to q_R all projections in views V'_1 and V'_2 , without compromising feasibility, and obtain the following query q'_R :

```
SELECT DISTINCT  $q'_R$ 
  V'_1.cid, V'_1.cpu, V'_1.ram, V'_1.price
  V'_2.cid, V'_2.cpu, V'_2.ram, V'_2.price
  V'_2.cid, V'_2.rate, V'_2.standard, V'_2.interface
FROM V'_1, V'_2
```

Hence, the maximum projection list of $q(n_9)$ consists of all projections in q'_R except $V'_1.\text{ram}$ and $V'_1.\text{price}$ which are mapped into from $q(n_9)$. The containment mappings are used here as well, so that the aliases in the maximum projection lists refer to aliases that appear in the current query. These lists are constructed in linear time.

Redundant Queries Removal Although the *Views Expansion* component inputs maximally-contained queries, not all syntactic extension queries it outputs are necessarily maximally-contained. It turns out that views expansion introduces redundancy across queries, i.e., expanded queries might contain one another. For example, if the current query is $q(n_1)$ in Figure II.5 (Snapshot 1 in Figure II.4), then MiniCon outputs two maximally-contained rewritings q_{R1} and q_{R2} over the views V'_1 and V'_2 which do not contain one another:

```
SELECT DISTINCT V'_1.ram, V'_1.price                                qR1
FROM V'_1

SELECT DISTINCT V'_2.ram, V'_2.price                                qR2
FROM V'_2
```

The expansion of q_{R1} though contains the expansion of q_{R2} , according to the definition of the views V'_1 and V'_2 in Example 4.

In order to preserve the rapid convergence and minimality guarantees of CLIDE (see Section II.D.1), we have to eliminate contained queries. This additional work is performed by the *Redundant Queries Removal* component, which we build from scratch and tests if one query is contained in another. The query containment test amounts to finding containment mappings between queries and is in general NP-complete in the query size. In practice, the constructed queries are small, and this test is very efficiently implemented [93]. We compute the containment mappings from query q_1 into query q_2 by constructing a canonical database [4] for q_2 , $canDB(q_2)$ and running q_1 over $canDB(q_2)$. To efficiently evaluate q_1 , we employ standard algebraic optimization techniques: we construct an algebraic operator tree for q_1 (left deep join tree), in which selections and projections are pushed and joins are implemented as hash joins.

The efficient implementation of the *Views Expansion* component proved crucial to the on-line response of CLIDE, since query containment tests are the bottleneck for CLIDE’s performance, as Section III.F demonstrates.

Redundant Actions Removal The output of the *Redundant Queries Removal* component is still not the set of minimal feasible extension queries FQ_{ME} that we are looking for, because they are not necessarily minimal extensions of the current query. For example, if q is the current query shown below, then q_E is the only feasible expansion query we get from MiniCon. q_E is not a minimal expansion query though. Query q_{ME} requires one action less than q_E to reach an equivalent query that minimally extends the current one.

```

SELECT DISTINCT Com1.ram, Com1.price                                 $q$ 
FROM Computers Com1, Computers Com2

SELECT DISTINCT Com1.ram, Com1.price                                 $q_E$ 
FROM Computers Com1, Computers Com2
WHERE Com1.cpu='P4' AND Com2.cpu='P4'

SELECT DISTINCT Com1.ram, Com1.price                                 $q_{ME}$ 
FROM Computers Com1, Computers Com2
WHERE Com1.cpu='P4'

```

The *Redundant Actions Removal* component finds FQ_{ME} by systematically detecting two identical constants that refer to identical columns of two tables with identical names but distinct aliases, dropping one of them at a time, and testing for equivalence with the initial query. The same rule is applied on self-joins.

III.F Experimental Evaluation

Our experimental evaluation shows that CLIDE is a viable on-line tool. The MiniCon algorithm was evaluated via extensive experiments in [69] to measure the time to find the maximally-contained rewritings of queries using views. The goal of our experiments was to show that the rest of the CLIDE components do not add a prohibitive cost, and that the algorithms of Sections III.B and III.C, as well

as our extensions and optimizations (efficient implementation of containment test, logging MiniCon’s containment mappings) are crucial in obtaining quick response times.

The Experimental Configuration To study how CLIDE scales with increasing complexity of the constructed query and with the number of views in the system, we used a synthetic experimental configuration, whose scaling parameters are K, L, M , as described below.

The schema. In the literature, synthetic queries are usually generated in one of two extreme shapes: chain queries and star queries. For a more realistic setting, we chose a schema which allowed us to build queries of a chain-of-stars shape, and in which joins follow foreign-key constraints (the most common reason for joins). To this end, we picked a schema comprised of a relation $A(ka, a)$ playing the role of a star center, which is linked (via foreign key constraints) to K relations $\{B_i(kb, fb, b)\}_{0 \leq i \leq K}$ (the star corners). Each B_i is in turn the center of another star whose L corners are given by the relations $\{C_{i,j}(kc, fc, c)\}_{0 \leq j \leq L}$. ka, kb, kc are respectively the key columns for A , the B_i ’s and the $C_{i,j}$ ’s. In each B_i , fb is a foreign key referencing ka from A . In each $C_{i,j}$, fc is a foreign key referencing kb from B_i .

The Views. The MiniCon experiments in [69] consider two extremes for view shapes, one very favorable, the other one leading to long rewriting time. The views in our configuration fall in the middle of this spectrum, and are more realistic. Each view we picked covers one of the foreign-key-based joins suggested by the schema. Moreover, we introduced selections with constants in these views, to force the interface to propose not only tables and joins, but also selections. For each i , we introduced M views $\{V_i^n\}_{0 \leq n \leq M}$ joining A with B_i and imposing a selection comparing the b column with some constant c_n . For each i, j we introduced M views $\{V_{i,j}^n\}_{0 \leq n \leq M}$ joining B_i with $C_{i,j}$ and comparing the c column to the constant c_n .

V_i^n : SELECT x.a, y.kb, y.b FROM A x, B _i y WHERE x.ka=y.fb AND y.b=c _n	$V_{i,j}^n$: SELECT y.kb, y.b, z.c FROM B _i y, C _{i,j} z WHERE y.kb=z.fc AND z.c=c _n
---------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------

There are $K \times M + K \times L \times M$ views in the configuration. For an intuitive interpretation of our abstract configuration, let the B_i tables stand for computer accessories, such as network cards, storage, keyboard, etc. For instance if B_1 plays the role of the `NetCards` table in Figure II.2 and A that of `Computers`, then the view V_1^3 provides the computers compatible with a network card satisfying a selection condition with constant c_3 .

The Queries. We scripted a family of interactions in which the simulated user starts by performing an A table action and then follows only blue and yellow suggestions, continuing even after reaching feasible queries.

After the initial A table action, CLIDE suggests joins with the B_i 's. If any of these suggestions are taken (say by picking B_p), CLIDE suggests the corresponding selections on B_p 's column b , as a list of options c_1, \dots, c_M . It also suggests table actions leading to the join of A with some other B_j or of B_p with some $C_{p,o}$. When the simulated user picks a selection with c_n it reaches a feasible query having a rewriting using V_p^n . When this feasible query is extended to join B_p with some $C_{p,o}$, CLIDE suggests (among others) selections comparing $C_{p,o}$'s column c to some constant. Picking one of these, say c_r , generates another feasible query, which has a rewriting that joins V_p^n with $V_{p,o}^r$.

The Measurements The measurements were conducted on a dedicated workstation (Pentium 4 3.2GHz, MS Windows XP Pro, 1GB of RAM) using Sun's JRE-1.5.0. All times measured are elapsed times.

We generated four configurations by fixing $K = 7$, $L = 3$ and varying $M = 4, 6, 8, 10$, yielding respectively 112, 168, 224 and 280 views. Figure III.3 reports the time CLIDE took to come up with the suggestions at each current query. Query (n, m) is a query reached after performing n table actions and joins, and m selections. On the horizontal axis, all odd-position queries are infeasible, while even-position queries are all feasible, being obtained by adding a relevant

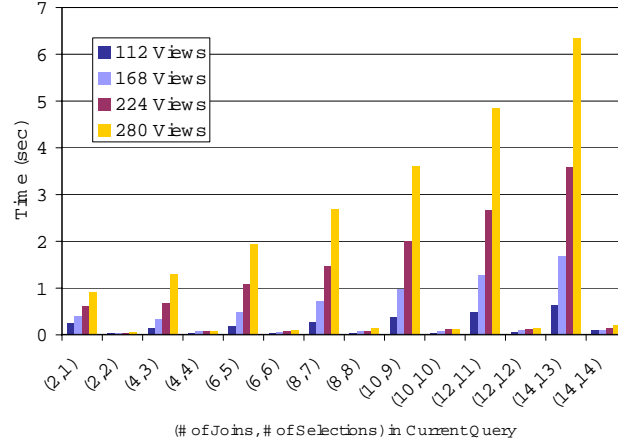


Figure III.3: CLIDE's response time

selection to their predecessor.

Notice that, while CLIDE's response is good overall, scaling to large number of views, it is much better for feasible queries. This is an expected result, since CLIDE needs to consider a single closest feasible query, i.e., the one that the user has reached, as opposed to the number of closest feasible queries when the current query is infeasible. The bottleneck in CLIDE's performance turns out to be the containment tests, which are a consequence of the views expansion. For instance, for query (14, 13), there are 700 expanded queries of which only 10 are non-redundant. These queries are quite sophisticated, joining up to 15 views. To identify them, CLIDE runs pairwise containment tests over the 700 redundant queries, then it minimizes the 10 queries invoking more containment tests. This work dominates the response time. 8311 containment tests need 6 seconds out of the 6.4 seconds of the elapsed time. The reason CLIDE scales to these query and view sizes is the efficient implementation of the containment test.

Note that when parameterized selections do not appear in the views, we could invoke MiniCon only when the user reaches a feasible query. We could exploit the fact that one interaction step along an edge $n \xrightarrow{a} n'$ changes $q(n)$ only incrementally. If a was a yellow or blue action, $FQ_C(n')$ would be contained in $FQ_C(n)$ and we would not need to call MiniCon to compute $FQ_C(n')$. Instead,

we could inspect the containment mappings from $q(n)$ into $FQ_C(n)$ and we could compute $FQ_C(n')$ by pruning those mappings that would not be consistent with action a and dropping from $FQ_C(n)$ all queries into which there would be no more containment mappings. This optimization would be in effect as long as the user would perform yellow and blue actions and for the periods of the interaction between feasible queries.

III.G Discussion and Related Work

Alternative query formulation paradigms have been proposed in the literature [76], but the QBE paradigm is the one that users are mostly familiar with today. As an alternative to a visual query builder, one could try to exploit existing formalisms for compact descriptions of infinite sets of supported queries. These focus mainly on sets of binding patterns [28, 51, 73, 94] and sets of parameterized queries described by the infinite unfoldings of recursive Datalog programs [50, 90]. However, these representations are meant for consumption by rewriting algorithms and not by humans: checking whether a given query is supported requires non-obvious rewriting algorithms, especially when the set of indirectly supported queries is enhanced via additional processing inside a mediator. This is a key obstacle to the practical utilization of current query rewriting algorithms for interactive query development, forcing the query writer into a trial-and-error loop.

There are many scenarios which would benefit from CLIDE's approach to query building. One example is the setting of [94], which is a special case of a service-oriented architecture with parameterized views restricted to identity views over individual tables. Their algorithm infers binding patterns for queries against these views, and could conceptually be used by the user to reach a feasible query by providing appropriate bindings. However, the user queries may be adorned with exponentially many binding patterns, turning the visual inspection by the user into a cumbersome process.

Another obvious CLIDE application is in data privacy enforcement. [74, 48] allow data owners to identify the non-sensitive data they are willing to export by means of parameterized, virtual views against the proprietary data. Data consumers formulate their queries against the proprietary database as well, but their queries are rejected [74] or return null values [48] if they are not feasible according to the virtual views. Again this leads to a frustrating trial-and-error development process.

The implementation of the CLIDE back-end described here requires as one building block an algorithm for finding maximally-contained rewritings. We picked MiniCon [69] because we had access to the code, but we could have swapped it with any other one, such as those in [28, 58]. Their applicability to our problem comes as a pleasant surprise, as the original goal of these algorithms is different: to provide an underestimate approximation of the query answer when the query is not feasible.

Other systems [58] automatically formulate and answer an overestimate or underestimate of the submitted query. We believe that in many applications the user needs full control and understanding of what she can ask and which precise query is being answered.

III.H Proofs

Proof of Lemma III.B.1: Recall that by definition every $q_1 \in FQ_{ME}(n)$ is feasible and contained in $q(n)$. q_1 is also closest to $q(n)$, because otherwise there would have to exist a feasible query q_2 , distinct from q_1 , and the paths $n \rightarrow n_2 \rightarrow n_1$ in the interaction graph, with $q_2 = q(n_2)$ and $q_1 = q(n_1)$. But then $q_1 \sqsubseteq q_2$ thus contradicting $q_1 \in FQ_{ME}(n)$. \diamond

Proof of Lemma III.B.2: Let $q_1 = q(n_1)$ where $q_1 \in FQ_C(n)$ and assume that there exists some node m reachable from n and a containment mapping h from $q(m)$ into $q(n_1)$. Let k be the number of table atoms in $q(m)$ minus the number

of table atoms in $q(n_1)$. Note that $0 \leq k$, otherwise $q(n_1)$ can be obtained by strictly extending $q(m)$ with table actions. But then n_1 is reachable from m in the interaction graph. Since m is reachable from n , we obtain the contradiction that $q(n_1)$ is not a closest query to $q(n)$.

Claim. We prove by induction on k that whenever there is a containment mapping from $q(m)$ into q_1 and m can be reached from n in the interaction graph, q_1 can be obtained from $q(m)$ by applying k alias collapse steps.

The Claim implies the lemma, since for each $q_1 \in FQ_C(n)$, there exists some $q(m) \in FQ_{ME}(n)$ and a containment mapping from $q(m)$ into q_1 .

The base case $k = 0$ is trivial. For the step, let $q(m)$ have $k + 1$ more table atoms than q_1 , and let h be the containment mapping from $q(m)$ into q_1 . Suppose that no table atoms of $q(m)$ have the same image under h . Then $q(n_1)$ has at least as many table atoms as $q(m)$ and $k + 1 < 0$, contradiction. Therefore there must exist Rr_1, Rr_2 in $q(m)$ which have the same image under h . Substitute r_1 for r_2 in $q(m)$ and drop the duplicate table atom to obtain a query q' . Notice that h remains a containment mapping from q' into q_1 . Also notice that q' is reachable from $q(n)$ and that q' now only has k more table atoms than q_1 . By induction hypothesis, q_1 is obtained by k alias collapse steps from q' , therefore by $k + 1$ collapse steps from $q(m)$. \diamond

Proof of Theorem 7: Follows from Lemmas III.B.1 and III.B.2. \diamond

Chapter IV

Data-Oriented Web Service Interfaces

This chapter presents the process of exporting query capabilities using the Query Set Specification Language (QSSL). Section IV.A provides the background for QSSL and introduces the running example for the rest of the chapter. Section IV.B formally defines QSSL and the supported query language. Moreover, it shows how QSSL accommodates recursive schemas and describes which properties can be verified. Section IV.C discusses possible extensions of QSSL and Section IV.D covers the related work.

IV.A Background

Web Services Description Language (WSDL) [15] provides an XML format for describing functions offered via web services. The function signatures typically have fixed numbers of input and output parameters. However, the “function” paradigm is not adequate when the software components behind the web services are databases. One typically associates one function with each parameterized query but this is problematic since databases often allow a large or even infinite set of parameterized queries over their schema. For example, the administrator of a product catalog database may want to allow any query that selects

products by a combination of selection conditions on the product's attributes. Assuming the product has, say, 10 attributes, it is obviously impractical to specify 2^{10} function signatures. In the particular example one can resolve the issue simply by allowing some input parameters to be *null*. This situation is generalized by QSSL to capture multiple function signatures in just one WSDL operation [15].

In addition, the function paradigm does not state explicitly either the relationship between the input parameters and the output or the semantic connections the available functions have with each other and with the underlying database. We classify such *web services* as *functional* and we argue that they are inappropriate for exporting structurally rich and functionally powerful information sources, such as relational and emerging XML databases.

We introduce a WSDL extension that enables *Data Services*, which overcome the limitations of functional web services. A Data Service exports the XML Schema [31] of an XML view. The Data Service also provides a set of parameterized queries that can be executed against the view. Hence the relationship between the input and output parameters is explicit, since the input corresponds to a query and the output to its result. Note that the view typically (but not necessarily) corresponds to a part of the underlying database.

Given an underlying database, the Query Set Specification Language (QSSL) allows the concise and semantically meaningful description of set of parameterized queries. The set may be very large or even infinite, since powerful information sources (such as relational databases) support a large number of parameterized queries. Consequently, QSSL must be able to describe sets of parameterized queries without requiring exhaustive enumeration of them.

QSSL concisely describes sets of tree pattern (subset of XPath) queries. It lends itself to a compact and intuitive visual notation that forms the basis of a GUI that could allow the authoring of Data Services.

Figure IV.1 shows the architecture of a Data Service published by a data source with a given view XML Schema. The query capabilities exported by a

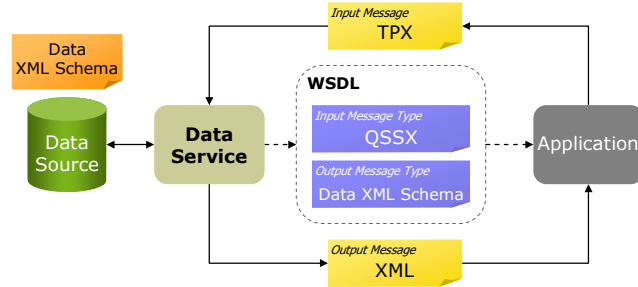


Figure IV.1: Data Service Architecture

Data Service are published as a WSDL-based web service [15] that provides an application with the means to formulate valid and acceptable queries and to be aware of the structure of the result. Notice that we translate QSSL specifications into XML Schemas and we are thus compatible with the WSDL specification. The Data Service receives an input message from the application and replies with an output message or a fault. The input message is a tree pattern (TP) query (subset of XPath), defined in Section IV.B, expressed in the TPX XML format, and the output message is an XML tree. The set of acceptable tree pattern queries (i.e., the set of acceptable input messages) is a QSSL specification, defined in Section IV.B. A QSSL specification describes the possibly infinite set of parameterized queries that are acceptable. A QSSL specification is translated into an XML Schema (QSSX) describing the acceptable TPX messages.

IV.A.1 Example

The running example is based on the XML Schema in Figure IV.2a that describes the structure of an airline database holding information about flights. The schema describes the flights carried out by one or more airline companies, where each flight has an origin and destination (`from` and `to` elements) and is scheduled at least once per week. In turn, each flight has one or more legs with a code, an origin and a destination and optionally the type of the aircraft used. Note that the schema of the actual airline database may be “richer” but we focus

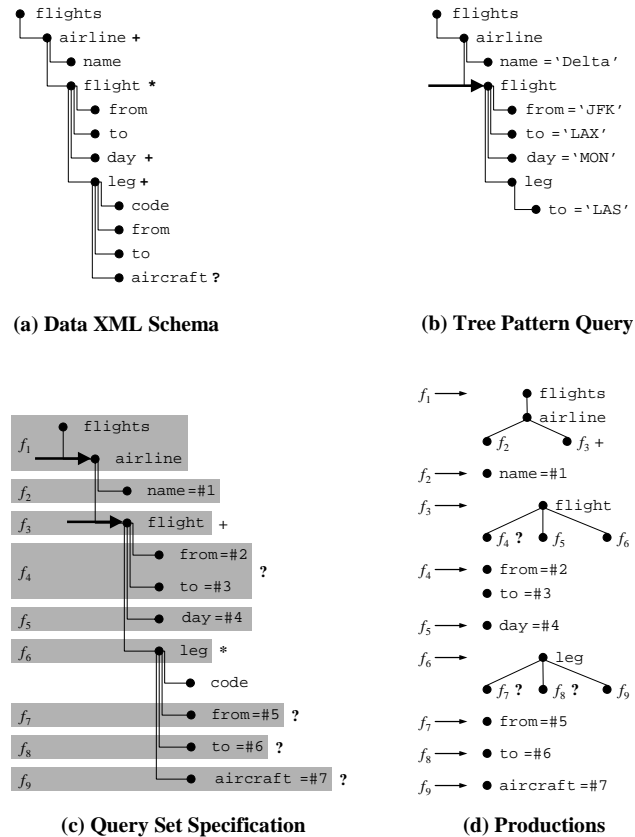


Figure IV.2: Airline Example

on the part that the database administrator exposes.

The database administrator allows queries that are having any combination of the following conditions:

- The name of the airline company is specified
- The origin and destination of one or more flights is optionally specified
- A day of the week is specified
- The origin of zero or more legs is optionally specified
- The destination of zero or more legs is optionally specified
- The aircraft used for zero or more legs is optionally specified

Notice that one may also specify combinations of origin, destination, and aircraft for legs. For the sake of the example, we also allow one to check whether a flight has a leg (existential condition).

The queries may return “airline” or “flight” elements.

The rest of the chapter presents the process of exporting such query capabilities using a Data Service.

IV.B Specifying Queries and Query Sets

We consider Data Services that support queries defined by a class of XPath expressions consisting of node tests, navigation along the child axis ‘/’ and the descendant axis ‘//’, and predicates denoted by ‘[]’. The established convention for representing this class of XPath expressions is to use *tree pattern (TP)* queries [7, 56]. We believe that support for tree pattern queries is a minimum Data Service requirement, since tree patterns are widely used in current applications, and since they are crucial building blocks of more expressive query languages such as XQuery [14]. Moreover, tree patterns provide an excellent visual paradigm which enables graphical user interfaces for constructing applications that produce and consume Data Services. For example, the XPath expression

```
flights/airline[name='Delta']/flight[from='JFK'][to='LAX'][day='MON'][leg[to='LAS']]
```

is represented by the tree pattern query in Figure IV.2b. The arrow pointing to the `flight` element node denotes the result node of the tree pattern query.

IV.B.1 QSSL Specifications

We define a Data Service by specifying the set of tree pattern queries it supports. First, we introduce *parameterized tree patterns (PTPs)*, which are TP queries where the constants are replaced with parameters. A PTP query specifies an infinite set of TP queries, each TP corresponding to a parameter instantiation. A Data Service exports a possibly infinite set of such parameterized tree pattern

queries. This set is succinctly encoded using a *QSSL specification*.

Definition 9 (*QSSL Specification*). A *QSSL specification* is defined as a 5-tuple $\langle F, \Sigma, P, S, R \rangle$, where:

- F is a finite set called the tree fragment names.
- Σ is a finite set, disjoint from F , called the element node names.
- P is a finite set of productions of general form $f \rightarrow tf_1, \dots, tf_n$ where $f \in F$ is a tree fragment name and each tf_i is a tree fragment. A tree fragment is a labeled tree consisting of:
 - Element nodes with labels from Σ . Leaf element nodes may be additionally labeled with a parameterized equality predicate of the form $= \#i$, where $\#i$ is a parameter and i is an integer.
 - Tree fragment nodes n labeled with a name $name(n) \in F$ and an occurrence constraint $occ(n) \in \{1, ?, +, *\}$. Tree fragment nodes can only appear as leaf nodes of a tree fragment. We often omit the occurrence constraint ‘1’.
 - Edges e either of child type, denoted by straight lines, or of descendant type, denoted by dashed lines.
- $S \in F$ is the start tree fragment name.
- $R \in \Sigma$ is a set called result node names. ◇

Example 10 The *QSSL specification* describing the airline Data Service from our motivating example is $A = \langle F, \Sigma, P, S, R \rangle$, where $F = \{f_1, \dots, f_9\}$, $\Sigma = \{flights, airline, name, flight, from, to, day, leg, aircraft\}$, P is the set of productions shown in Figure IV.2d, $S = f_1$ and $R = \{airline, flight\}$. ◇

A compact visual representation of this *QSSL specification* is given in Figure IV.2c, where tree fragments are depicted by shaded boxes with occurrence

constraints to their right. This visual representation could be the basis of a GUI for authoring QSSL specifications by displaying the XML Schema and using drag-and-drop actions.

Given the similarity between QSSL specifications and extended context-free grammars [6], we define the set of parameterized tree pattern queries described by a QSSL specification analogously to the language generated by a grammar. A QSSL specification defines the set of PTPs whose result node is in R and whose pattern is yielded by a sequence of *derivation steps* starting from the start fragment name S . At any step, given a tree fragment node n , the derivation step replaces n with the tree fragments on the right hand side of a production that has n on the left hand side. Depending on the occurrence constraint labeling n , the derivation step might replace it more than once or not at all. More specifically, if $occ(n) = 1$, then n is replaced with the corresponding tree fragments exactly once, and they all become children of n 's parent. If $occ(n) = ?$, then n is nondeterministically either deleted or relabeled with $occ(n) = 1$ before replacement. If $occ(n) = +$, then for a nondeterministically chosen $k \geq 1$, n is replaced with k copies of n , all siblings, with occurrence labels set to 1. If $occ(n) = *$, then n is labeled nondeterministically with $occ(n) = ?$ or $occ(n) = +$ first. The parameters introduced in every step are freshly renamed such that their name is unique across the tree fragment obtained so far.

A TP query is *accepted* by a QSSL specification A if and only if it corresponds to an instantiation of the parameters of a PTP query from the set defined by A . We denote with $TP(A)$ the set of TP queries accepted by A .

Example 11 *Figure IV.3 shows the sequence of derivations steps, denoted by the \Rightarrow symbol, that obtains the corresponding PTP query pq of the TP query q in Figure IV.2b. Note how the third derivation step replaces f_4 with the corresponding tree fragments, and how the fourth derivation step deletes f_7 and f_9 . After the final derivation step, the node labeled with the flight result node name is chosen, thus forming a PTP query pq . When pq 's parameters $[\#1, \#2, \#3, \#4, \#5]$ are*

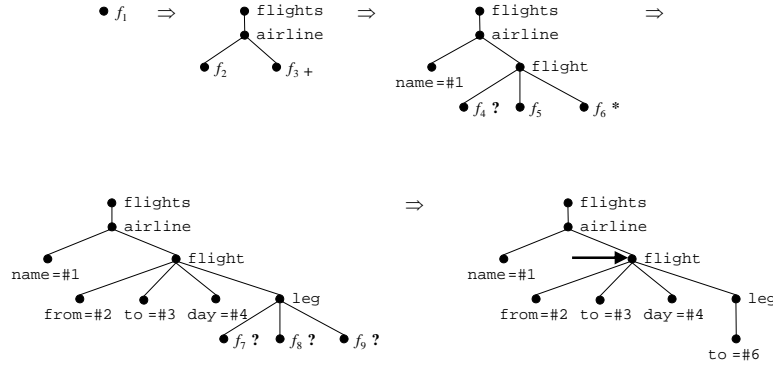


Figure IV.3: Example Derivation

instantiated with the constants [*Delta*, *JFK*, *LAX*, *MON*, *LAS*], we obtain the TP query q from Figure IV.2b. Therefore, q is accepted by A . \diamond

When the XML Schema is recursive, it describes documents of arbitrary depth. On these documents, there are TP queries of arbitrary pattern height with non-empty answer and it makes sense to export them in a Data Service.

Despite its fixed size (determined by the XML schema), a QSSL specification can represent such arbitrarily deep TP queries.

Example 12 *The recursive XML Schema in Figure IV.4a captures the structure of a family tree. Figure IV.4b shows a TP query that returns the persons found at any depth that are named “Kevin” and were born in “NY” such that at least one of his descendants is married to a person also named “Kevin” and also born in “NY”. Recall that the dotted lines in Figure IV.4b denote descendant edges.*

A QSSL specification that accepts, among others, the corresponding PTP query of the TP query in Figure IV.4b is shown in Figure IV.4c. Note the last node, labeled with the tree fragment name f_2 , representing the recursion in the schema. Formally, the above QSSL specification is defined as $B = \langle F, \Sigma, P, S, R \rangle$, where $F = \{f_1, \dots, f_8\}$, $\Sigma = \{\text{familyTree, person, name, place, spouse, children}\}$, P is the set of productions shown in Figure IV.4d, $S = f_1$ and $R = \{\text{person}\}$. Note how the recursion in productions f_2 , f_5 and f_8 allows for derivations of arbitrary length. It is easy to see that the PTP query corresponding to the TP query in

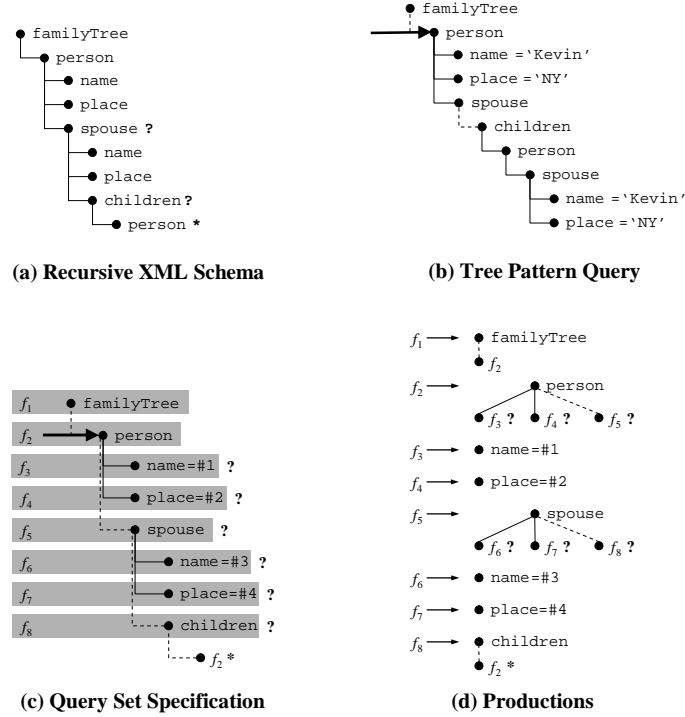


Figure IV.4: Family Tree Recursive Example

Figure IV.4b is obtained by a sequence of derivation steps using the production associated to f_2 twice. \diamond

IV.B.2 Reasoning about Data Services

Aside from facilitating the development of applications that are clients of the Data Service, QSSL specifications allow reasoning about Data Services. Below are examples of Data Service properties we would like to verify.

- *Membership of a query in a Data Service.* The most basic problem is to check if a client TP query q is accepted by a Data Service described by a QSSL specification A , i.e. $q \in \text{TP}(A)$.
- *Subsumption of Data Services.* Given services described by QSSL specifications A_1 and A_2 , check if $\text{TP}(A_1) \subseteq \text{TP}(A_2)$.
- *Totality of a Data Service.* Does the Data Service described by QSSL speci-

fication A accept all possible TP queries?

- *Overlap of Data Services.* Given services described by QSSL specifications A_1 and A_2 , check if $TP(A_1) \cap TP(A_2) \neq \emptyset$.

Of course, revisiting the analogy between QSSL specifications and extended context-free grammars, we could reduce these problems to decision problems on grammars. However, while the membership problem can be solved in this way, the other problems in the list reduce to well-known problems that are undecidable even for standard context-free grammars. Fortunately, it turns out that a QSSL specification can be translated to an equivalent top-down nondeterministic unranked tree automaton [16] (the translation is straightforward). QSSL specifications therefore describe regular tree languages, for which all problems listed above are decidable [16]. This observation should not come as a surprise given the similar result stating that DTDs, who look strikingly similar to extended context-free grammars, actually describe regular tree languages [79].

A practically important question is whether a client query can be answered using a finite subset of the queries described by a QSSL specification. This is related to the problem of answering queries using limited query capabilities [89].

IV.B.3 WSDL and XML Syntax

Our proposal for specifying Data Services is compatible with the standard Web Service Specification Language WSDL [15] in the sense that any QSSL specification can be translated into a WSDL specification.

In general, a WSDL specification describes the format of the messages that a service sends or receives using element declarations and type definitions drawn from the XML Schema type system [31]. A WSDL specifies many additional communication details, such as synchronicity, how sets of messages are grouped into one operation, etc., all of which are orthogonal to our proposal. A WSDL specification describing a Data Service restricts a general WSDL specification in

several ways, since the communication between the application and the Data Service is always synchronous and is carried out in a request/response fashion [77]. The input message represents the query received from the service, and the output message the result sent from the service. Both message types are described using the XML Schema type system.

A QSSL specification can be automatically translated into a WSDL specification using the well-known fact that XML Schemas describe regular tree languages themselves. We omit the details of the translation algorithm, but illustrate on an example. This example makes a convincing case for presenting users with a concise and visually intuitive representation such as a QSSL specification instead of the less readable XML syntax of the WSDL.

Example 13 *Appendix A shows the WSDL for the QSSL specification from Figure IV.2c. The schemas that describe the input and the output messages of the Data Service are imported in the beginning of the specification. The first schema describes the parameterized queries supported by the Data Service. A QSSL specification is expressed in XML Schema format (QSSX) in order to be contained in a WSDL specification. QSSX is an XML Schema that acceptable TPX queries conform to. The QSSX syntax for the QSSL specification in Figure IV.2c is shown in Appendix B. The second schema reveals the structure of the underlying database and presents a choice group consisting of the result node names in the result node names set of a QSSL specification. Appendix C shows the XML Schema for the QSSL specification in Figure IV.2c. As in the case of QSSL and QSSX syntax, TP queries need to be expressible in XML format in order to be contained in messages described by a WSDL specification. The XML syntax of TP queries, called TPX, is a subset of XQueryX [54], the XML syntax of XQuery. The TPX query equivalent to the TP query in Figure IV.2b is given in Appendix D. The XML Schema that defines the TPX language is presented in Appendix E. \diamond*

IV.C QSSL Extensions

QSSL can be enhanced to describe subsets of XQuery expressions beyond XPath ones, as well as additional constraints that restrict the co-occurrence of tree fragments.

In Figure IV.4c, for example, the QSSL specification only indicates that a parameterized equality predicate on the name and on the birth place of a person can optionally be part of an acceptable PTP query. The QSSL specification does not have the ability to succinctly express that these two predicates are mutually exclusive, or express that at least one of them must be part of an acceptable PTP query. It can achieve the desired effect by explicitly listing all acceptable combinations, but this defeats the purpose of QSSL.

In order to express these constraints, QSSL can be enriched with a set of *replacement constraints* including *atLeast*, *atMost* and *xor*.

For example, *atLeast*(1, { f_3 , f_4 }) expresses that in a derivation at least one of f_3 and f_4 must be replaced. *xor*({ f_3 , f_4 }, { f_6 , f_7 }) expresses that either the parameterized predicates on the name and on the place of a person or on the name and on the place of a spouse are part of an acceptable PTP query, but not both.

IV.D Related Work

In the past, the database community has conducted research on the related problems of answering queries using views [42], capability-based query rewriting [41, 89] and computation of query capabilities [94]. One approach assumes that a source exports a relational view with n attributes, and query capabilities are described as *binding patterns* [42]. Each binding pattern attaches a b (bound) or an f (free) adornment on each attribute of the exported view. Adornment b means that a value for the attribute is required in a query, while f means that a value is optional. The set of adornments can be enriched adding u , where a value for an attribute is not permitted, $c[s]$, where a value for an attribute is required and must

be chosen from the set of constants s , and $o[s]$, where a value in s is optional [94]. Note that each binding pattern defines a query template. Query capabilities described as binding patterns are characterized as *negative*, because they restrict the set of all possible queries against the exported view. Wrappers exporting binding patterns are called *thin*, because of their limited functionality to execute the input query against the underlying source.

Another approach describes sets of (parameterized) queries using the expansions of a Datalog program [50, 89]. In this work, it is shown that Datalog is not enough to cover even all yes/no conjunctive queries over a schema. It consequently showed that the RQDL extension can describe large sets, such as the set of all conjunctive queries over a schema. QSSL and Data Services also attempt to describe the capabilities of sources that support large sets of queries and aim to fuel the research on the problems considered in [41, 42, 89, 94] for the XML data model and the XQuery language [14].

On the industrial level, the effort is focused on turning relational database systems to web services providers by exporting data definition and manipulation operations via web services. These operations are either fixed or parameterized queries expressed in SQL or SQL/XML [29], stored procedures, or functions. Typically, a web service exporting a fixed query takes as input the name of the database operation, and possibly a parameter instantiation, and outputs either an XML document or a serialized object in a given programming language. No schema information of the underlying database is given, either for the input or the output. A list of systems implementing this architecture includes IBM's Document Access Definition Extension (DADx) for DB2 [44], Oracle's Database Web Services specification [55], Microsoft's SQL Server 2000 Web Services Toolkit [85] and BEA's WebLogic Workshop [91]. There is also an effort on consuming web services within the SQL query language, thus integrating relational data with web services.

Finally, the W3C Web Services Description Working Group [77] describes usage scenarios that focus on various types of communication using messages and

demonstrate how they can be carried out using web services. The technical issues focus on the direction of communication, i.e., request-response, solicit-response or one-way, whether a web service is synchronous or asynchronous and whether it supports conversations, rather than what query capabilities a database exports.

Chapter V

Graphical Query Interfaces for Semistructured Data

This chapter presents an overview of the QURSED system. In particular, Section V.A outlines the background of semistructured data and web-based graphical query interfaces, and reveals the need for the QURSED system. Section V.B provides an overview of the system architecture. Related systems, industrial and academic, and the contributions of QURSED are discussed in Section V.C. Section V.D defines the data model and Section V.E introduces the QURSED-generated running example for subsequent chapters and describes the end-user experience.

V.A Background

XML is a simple and powerful data exchange and representation language, largely due to its self-describing nature. Its advantages are especially strong in the case of semistructured data, i.e., data whose structure is not rigid and is characterized by nesting, optional fields, and high variability of the structure. An example is a catalog for complicated products such as sensors: they are often nested into manufacturer categories and each product of a sensor manufacturer comes with its own variations. For example, some sensors are rectangular and

have height and width, and others are cylindrical and have diameter and barrel style. Some sensors have one or more protection ratings, while others have none. The relational data model is cumbersome in modeling such semistructured data because of its rigid tabular structure.

The database community perceived the relational model's limitations early on and responded with labeled graph data models [3] that evolved into XML-based data models [32]. XML query languages (with most notable the emerging XQuery standard [14]), XML databases [78] and mediators [19, 26, 34, 52, 83] have been designed and developed. They materialize the in-principle advantages of XML in representing and querying semistructured data. Indeed, mediators allow one to export XML views of data found in relational databases [34, 83], XHTML pages, and other information sources, and to obtain XML's advantages even when one starts with non-XML legacy data. QURSED automates the construction of web-based query forms and reports for querying semistructured, XML data.

Web-based query forms and reports are an important aspect of real-world database systems [11, 84] - albeit semi-neglected by the database research community. They allow millions of web users to selectively view the information of underlying sources. A number of tools [88, 21, 45] facilitate the development of web-based query forms and reports that access relational databases. However, these tools are tied to the relational model, which limits the resulting user experience and impedes the developer in his efforts to quickly and cleanly produce web-based query forms and reports. QURSED is, to the best of our knowledge, the first web-based query forms and reports generator with focus on semistructured XML data.

QURSED produces query form and report pages that are called *QFRs*. A *QFR* is associated with a *Query Set Specification (QSS)*. A *QSS* describes formally the complex query and reporting capabilities [90] of a *QFR*. These capabilities include the large number of queries that a form can generate to the underlying XML query processor and the different structure and content of the query result.

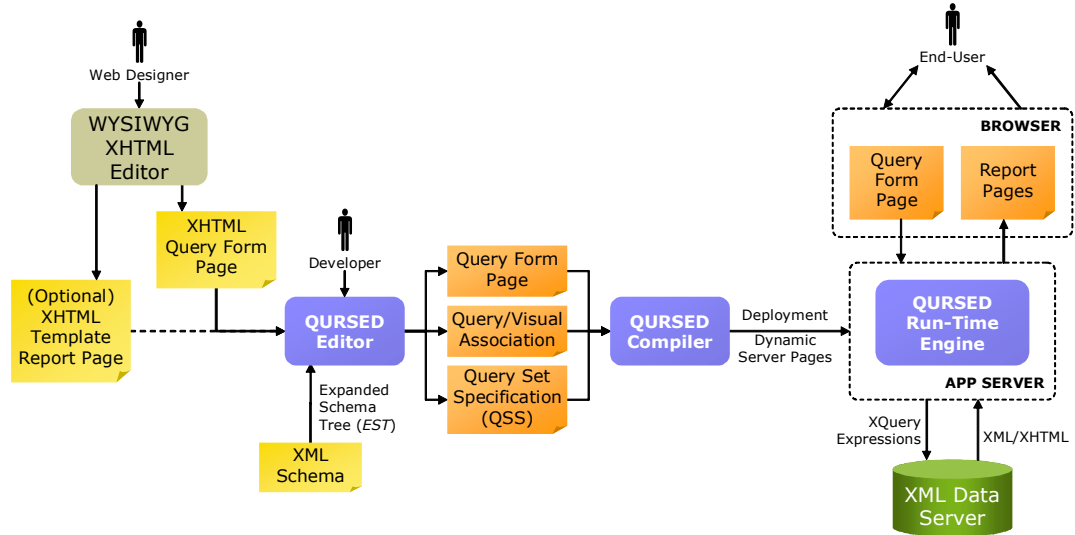


Figure V.1: The QURSED System Architecture

The emitted queries are expressed in XQuery and the query results are expressed directly in XHTML that renders the report page.

V.B System Overview and Architecture

We discuss next the QURSED system architecture, shown in Figure V.1, the process and the actions involved in producing a *QFR*, and the process by which a *QFR* interacts with the end-user, emits a query, and displays the result. We also introduce terms used in the rest of the chapter and in subsequent ones. QURSED consists of the *QURSED Editor*, which is the design-time component, the *QURSED Compiler*, and the *QURSED Run Time Engine*.

The Editor inputs the XML Schema that describes the structure of the XML data to be queried and constructs an *Expanded Schema Tree (EST)* out of it. The *EST* is a structure that serves as the basis for building the query set specification and is a visual abstraction of the XML Schema that the developer interacts with. The Editor also inputs an *XHTML query form page* that provides the static part of the form page, including the XHTML form controls [71], such as `select` (“drop-down menus”) and `text` (“fill-in-the-box”) input controls, that the

end-user will be interacting with. It may additionally input an optional *template report page* that provides the XHTML structure of the report page. In particular, it depicts the nested tables and other components of the page. It is just a template, since we may not know in advance how many rows/tuples appear in each table. The query form and template report pages are typically developed with an external “What You See Is What You Get” (WYSIWYG) editor, such as [43]. If a template report page is not provided, the developer can automatically build one using the Editor.

The Editor displays the *EST* and the XHTML pages to the developer, who uses them to build the query set specification of the *QFR* and the query/visual association. The *QSS* focuses on the query capabilities of the *QFR* and describes the set of queries that the form may emit. The query description is based on the formalism of the *Tree Query Language (TQL)* described in Chapter VI. The *QSS*’s key components are the parameterized *condition fragments*, the fragment *dependencies* and the *result tree generator*. Each condition fragment stands for a set of conditions (typically navigations, selections and joins) that contain *parameters*. The query/visual association indicates how each parameter is associated with corresponding *XHTML form controls* of the query form page. The form controls that are associated with the parameters contained in a condition fragment constitute its *visual fragment*. Dependencies can be established between condition fragments and between the values of parameters and fragments, and provide fine-grained control on what queries can be submitted and which visual fragments are eligible to appear on the query form page at each point (see Figure VI.8 in Section VI.D). Finally, the result tree generator specifies how the source data instantiate and populate the XHTML template report page.

The *QURSED Compiler* takes as input the output of the Editor and produces *dynamic server pages*, which control the interaction with the end-user. Dynamic server pages are implemented in QURSED as [60], while Active Server Pages [9] is another possible option. The dynamic server pages, the query set

specification and the query/visual association are inputs to the *QURSED Run-time Engine*. In particular, the dynamic server pages enforce the dependencies between the visual fragments on the query form page and handle the navigation on the report page. The engine, based on the query set specification and the query/visual association, generates an XQuery expression when the end-user clicks “Execute”, which is sent to the XML Data Server and its XHTML result is displayed on the report page.

V.C Related Work

The QURSED system relates to four wide classes of systems, coming from both academia and industry:

1. *Web-based Form and Report Generators*, such as [88], [21], and [45]. All of the above enable the development of web-based applications that create form and report pages that access relational databases, with the exception of XQForms [67], which targets XML data. QURSED is classified in the same category, except for its focus on semistructured data.
2. *Visual Querying Interfaces*, such as QBE [95] and Microsoft’s Query Builder [45], which target relational databases, and XML-GL [22], EquiX [20], BBQ [57], VQBD [18], the Lorel’s DataGuide-driven GUI [40], and PESTO [17], which target XML or object-oriented databases.
3. *Schema Mapping Tools*, such as IBM’s Clio [68], Microsoft’s BizTalk Mapper [80], [86] and BEA’s Data View Builder [35]. These are graphical user interfaces that facilitate the data transformation from one or more source XML Schemas to a target XML Schema. The user constructs complex XQuery [14] or XSLT [46] expressions through a set of visual actions. These tools are mainly used in integration scenarios.
4. *Data-Intensive Web Site and Application Generators*, such as Autoweb [37],

Araneus [10], Strudel [33] and Application Manifold [30]. These are recent research projects proposing new methods of generating web sites, which are heavily based on database content. An additional extensive discussion on this class of systems can be found in [36].

Web-based Form and Report Generators create web-based interfaces that access relational databases. Popular examples are [88], [21], and [45]. The developer uses a set of wizards to visually explore the tables and views defined in a relational database schema and selects the one(s) she wants to query using a query form page. By dragging and dropping the attributes of the desired table to XHTML form controls [71] on the page, she creates conditions that, during runtime, restrict the attribute values based on the end-user's input. The developer can also select the tables or views to present on a report page, and by dragging and dropping the desired attributes to XHTML elements on the page, e.g., table cells, the corresponding attribute values will be shown as the element's content. The developer also specifies the XHTML region that will be repeated for each record found in the table, e.g., one table row per record. These actions are translated to scripting code or a set of custom XHTML tags that these products generate. The custom tags incorporate common database and programming languages functionality and one may think of them as a way of folding a programming/scripting language into XHTML. The three most popular custom tag libraries today are [60], Active Server Pages [9] and Macromedia ColdFusion Markup Language [21].

Those tools are excellent when flat uniform relational tables need to be displayed. The visual query formulation paradigm offered to the developer allows the expression of projections, sort-bys, and simple conditions. However, the development of form and report pages that query and display semistructured data requires substantial programming effort.

Visual Querying Interfaces are applications that allow the exploration of the schema and/or content of the underlying database and the formulation of queries. Typical examples are the Query-By-Example (QBE) [95] interface and

Microsoft’s Query Builder [45], which target the querying of relational databases. Recent visual front-ends such as XML-GL [22], EquiX [20], BBQ [57], VQBD [18], the Lorel’s DataGuide-driven GUI [40], and PESTO [17] target the querying of XML and object-oriented databases. Unlike the form and report generators, which produce web front-ends for the “general public”, visual querying interfaces present the schema of the underlying database to experienced users, who are often developers building a query, help them formulate queries visually, and display the result in a default fashion. The user has to, at the very least, understand what the meaning of “schema” is and what the model of the underlying object structure is, in order to be able to formulate a query. For example, the QBE user has to understand what a relational schema is and the user of Lorel’s DataGuide GUI has to understand that the tree-like structure displayed is the structure of the underlying XML objects. These systems have heavily influenced the design of the Editor because they provide an excellent visual paradigm for the formulation of fairly complex queries.

In particular, EquiX allows the visual development of complex XML queries that include quantification, negation and aggregation. EquiX and BBQ use some form of the *EST* and of the corresponding visual concept, but they still require basic knowledge of query language primitives. Simple predicates, Boolean expressions and variables can be typed at terminal nodes and quantifiers can be applied to non-terminal nodes. In a QBE-like manner, the user can select which elements of the DTD to “print” in the output but the XML structure of the query result conforms to the XML structure of the source, i.e., there is no restructuring ability.

A more powerful visual query language is XML-GL that uniformly expresses XML documents, DTDs and queries as graphs. Queries consist of a set of extraction query graphs, a set of construction query graphs, and a set of bindings from nodes of one side to nodes of the other. In terms of expressiveness, XML-GL is more powerful than BBQ and EquiX, because of its ability to construct com-

plex results using grouping, aggregate and arithmetic functions. It also supports heterogeneous union, in a fashion similar to TQL. XML-GL is less powerful than XQuery though, since recursive queries are not expressible and nested subqueries are partially supported. An advantage of XML-GL is that it can be implemented as a visual front-end to an XQuery processor, since the correspondence between their semantics is straightforward. The disadvantage of XML-GL is that it doesn't make the common case easy. The interface is not intuitive for simple queries until the developer gets familiar with the visual semantics of the language.

It is important to note that the described visual query formulation tools and the Editor have very different goals: The goal of the former is the development of a query or a query template by a database programmer, who is familiar with database models and languages. The goal of the latter is the construction from an average web developer of a form that represents and can generate a large number of possible queries.

Schema Mapping Tools are graphical user interfaces that declaratively transform data between XML Schemas in the context of integration applications. IBM's Clio [68], Microsoft's BizTalk Mapper [80], [86] and BEA's Data View Builder [35] are representative examples. The transformation is a three-step process that is based on multiple source XML Schemas and a single target XML Schema that are visualized and presented to user. The first step discovers and creates correspondences between one or more elements of the source schemas and a single target element without attaching any specific semantics to them. The second step turns correspondences to mappings by specifying exactly how the source elements are transformed to the target element. Selection predicates, inner and outer joins, arithmetic, string and user defined functions are a few examples of the supported functionality. Clio [68] goes one step further and explains the difference between different mappings interactively by giving examples to the user based on small datasets. The third step of the transformation process generates either an XQuery [14] or an XSLT [46] expression that actually implements the

transformation.

Note that the first two steps above are carried out using visual actions only, so the user does not need to be aware of the particular query language used by each tool. These visual actions greatly facilitate data integration by simplifying the transformation process, especially when someone takes into account that the generated query expressions are particularly complex and hard to write by hand.

QURSED's Editor adopts part of the functionality provided by the schema mapping tools for a different purpose. More specifically, the Editor creates two types of transformations without making a distinction between correspondences and mappings. First, it creates query/visual associations that map form controls on the XHTML query form page to parameters of selection predicates, in order to generate queries that filter the data. And second, it creates a transformation between a single XML Schema and an XHTML template report page in order to construct the report pages.

Data-Intensive Web Site and Application Generators. Autoweb [37], Araneus [10] and Strudel [33] are excellent examples of the ongoing research on how to design and develop web sites heavily dependent on database content. All of them offer a data model, a navigation model and a presentation model. They provide important lessons on how to decouple the query aspects of web development from the presentation ones. (Decoupling the query from the presentation aspects is an area where commercial web-based form and report generators suffer.) Strudel is based on labeled directed graphs model for both data and web sites and is very close to the XML model of QURSED.

The query language of Strudel, called StruQL, is used to define the way data are integrated from multiple sources (data graph), the pages that make up the web site, and the way they are linked (site graph). Each node of the site graph corresponds to exactly one query, which is manually constructed. Query forms are defined on the edges of the site graph by specifying a set of free variables in the query, which are instantiated when the page is requested, producing the end node

of the edge. Similarly, Autoweb and Araneus perceive query forms as a single query, in the sense that the number of conditions and the output structure are fixed. In Strudel, if conditions need to be added or the output structure to change, a new query has to be constructed and a new node added to the site graph. In other words, every possible query and output structure has to be written and added to the site graph. QURSED is complementary to these systems, as it addresses the problem of encoding a large number of queries in a single *QFR* and also of grouping and representing different reports using a single site graph node.

Application Manifold [30] is the first attempt to expand a data integration framework to an application integration one. The system is capable of generating web-based e-commerce applications by integrating and customizing existing ones. Applications' flow is modeled and visually represented using UML State Charts that consist of states, corresponding to web pages that provide activities, linked by transitions, corresponding to navigation links that the end user can follow, and containing actions, corresponding to method calls that trigger other transitions and/or alter the application's state. Application integration and customization is specified using a declarative language that allows for optimization and verification of the generated application.

Related to QURSED is also prior work on capability-description languages and their use in mediator systems [49, 90]. The *QSS* formalism of QURSED is essentially a capability description language for query forms and reports over XML data. The prior work on capabilities has focused on describing the capabilities of query *processors* with an underlying relational data model. Instead the *QSS* captures the complex query and reporting capabilities of query *forms* over semistructured data.

There is also the prior work of the author on the XQForms system that declaratively generates Web-based query forms and reports that construct XQuery expressions [67]. This work introduces a software architecture that allows an extensible set of XHTML input controls to be associated with element definitions of

an XML schema via an annotation on the XML Schema. It also presents different “hard-wired” ways the system provides for customizing the appearance of reports. The set of queries produced by the system are conjunctive and its spectrum is narrow because of the limitations of the XML Schema-based annotation. This work does not describe how the system encodes or composes queries and results of queries based on end-user actions.

Finally, there is the XForms W3C standard [27], which promotes the use of XML structured documents for communicating to the web server the results of the end-user’s actions on various kinds of forms. XForms also tries to provide constructs that change the appearance of the form page on the client side, without the need of coding. QURSED can use these constructs for the evaluation of dependencies, thus simplifying the implementation.

V.C.1 Novel Contributions of QURSED

Forms and Reports for Semistructured Data. QURSED generates form and report pages that target the needs of interacting with and presenting semi-structured data. Multiple features contribute in this direction:

1. QURSED generates queries that handle the structural variance and irregularities of the source data by employing appropriate forms of disjunction. For example, consider a sensor query form that allows the end-user to check whether the sensor fits within an envelope with length X and width Y , where X and Y are end-user-provided parameters. The corresponding query has to take into consideration whether the sensor is cylindrical or rectangular, since X and Y have to be compared against a different set of dimension attributes in each case.
2. Condition fragment dependencies control what the end-user can ask at every point. For example, consider another version of the sensor query form that contains a selection menu where the end-user can specify whether he is in-

terested in cylindrical or rectangular sensors. Once this is known, the form transforms itself to display conditions (e.g., diameter) that pertain to cylindrical sensors only or conditions (e.g., height and width) that pertain to rectangular sensors only.

3. On the report side, data can be automatically nested according to the nesting proposed by the source schema or can be made to fit XHTML tables that have variance in their structure and different nesting patterns. Structural variance on the report page is tackled by producing heterogeneous rows/tuples in the resulting XHTML tables.

Loose Coupling of Query and Visual Aspects. QURSED separates the logical aspects of query forms and reports generation, i.e., the query form capabilities, from the presentation aspects, hence making it easier to develop and maintain the resulting form and report pages. The visual component of the forms can be prepared with any XHTML editor. Then the developer can focus on the logical aspects of the forms and reports: Which are the condition fragments? What are their dependencies? How should the report be nested? The coupling between the logical and the visual part is loose, simple, and easy to build: The query parameters are associated with XHTML form controls, the condition fragments are associated with sets of XHTML form controls, and the grouped elements (see Chapter VI) of the result tree are associated with the nested tables of the report.

Powerful and Succinct Description Language for Query Form Capabilities. We provide formal syntax and semantics for the *QFR* query set specifications, which describe query form capabilities by succinctly encoding large numbers of meaningful semistructured queries. The specifications primarily consist of parameterized condition fragments and dependencies. The combinations of the fragments lead to large numbers of parameterized queries, while the dependencies guarantee that the produced queries make sense given the XML Schema and the semantics of the data.

The query set specifications are using the Tree Query Language (TQL), which is a calculus-based language. TQL is designed to handle the structural variance and missing fields of semistructured data. Nevertheless, TQL’s purpose is not to be yet another general-purpose semistructured query language. Its design goals are to:

1. facilitate the definition of query set specifications and, in particular, of condition fragments, and
2. provide a tree-based query model that captures easily the schema-driven generation of query conditions by the forms component of the Editor and also maps well to the model of nested tables used by the reports.

XML, XHTML, and XQuery-Based Architecture. The QURSED architecture and implementation fully utilizes XQuery and the interplay of XML/XHTML. The result is an efficient overall system, when compared either against relational-based front-end generators or against conventional XML-based front-end architectures, such as [92]. An XML-related efficiency is derived by the fact that XML is used throughout QURSED: XML is the data model of the source on which XML queries, in XQuery syntax, are evaluated, and is also used to deliver the presentation - in the form of XHTML. The elimination of internal model mismatches yields significant advantages in the engineering and maintainability of the system.

V.D Data Model, XML Schema and Expanded Schema Tree

QURSED models XML data as labeled ordered tree objects (*lotos*), such as the sample data set shown in Figure V.2a that describes two proximity sensor products. Each internal node of the labeled ordered tree represents an XML element and is labeled with the element’s tag name. The list of children of a node represents the sequence of elements that make up the content of the element. A

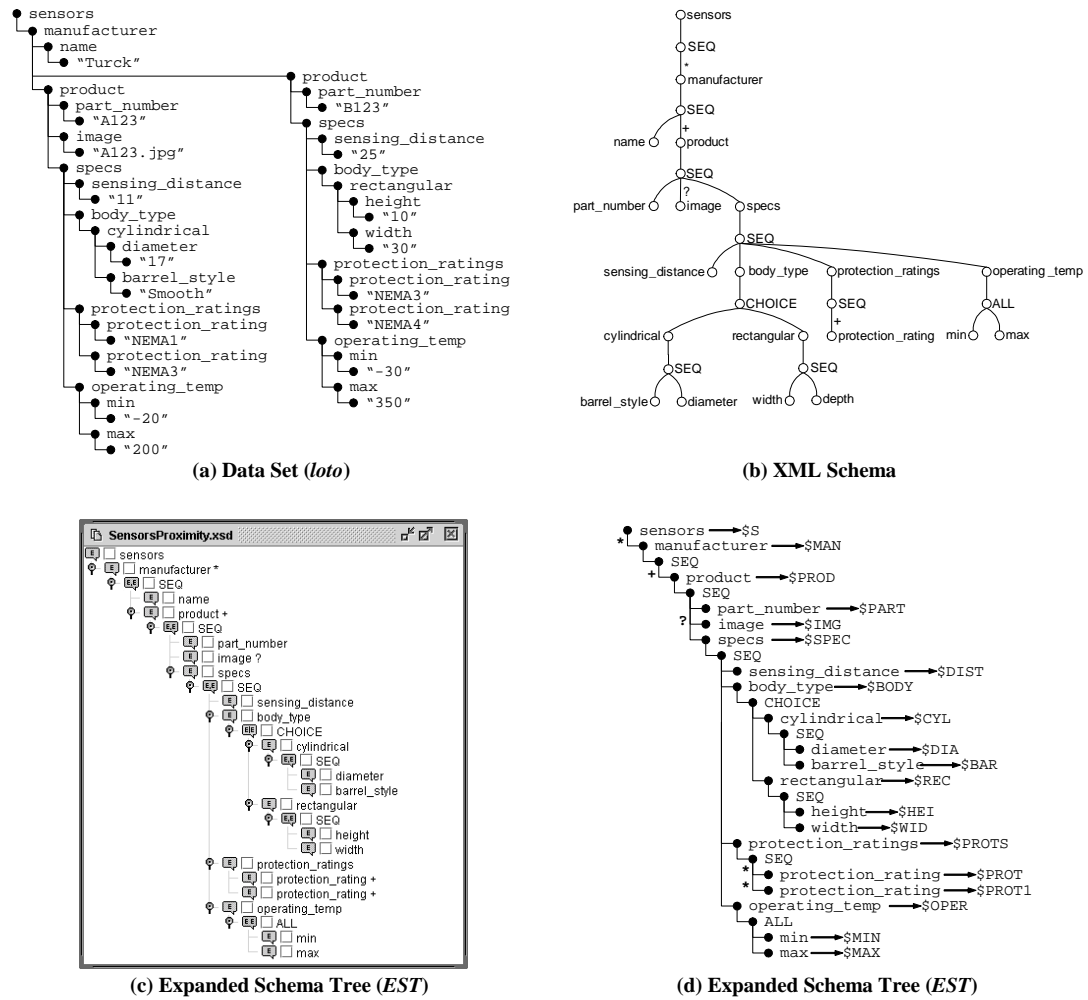


Figure V.2: Example Data Set, XML Schema and Expanded Schema Tree

leaf node holds the string value of its parent node. If n is a node of a *loto*, we denote as $tree(n)$ the subtree rooted at n .

In the sample data set of Figure V.2a, the top `sensors` node contains a `manufacturer` node, whose `name` is “Turck”. This manufacturer contains a list of two `product` nodes, whose direct subelements contain the basic information of each sensor. The first sensor’s `part_number` is “A123” and has an `image`, while the second’s one is “B123” and has no image. The technical specification of each sensor is modeled by the `specs` node, whose content is quite irregular. For example, the `body_type` of the first sensor is `cylindrical`, and has `diameter` and `barrel_style`, while the second one is `rectangular` and has `height` and `width`. Also, both sensors have more than one `protection_rating` nodes and have `min` and `max` operating temperature.

The XML Schema that describes the structure of the sample data set of Figure V.2a is shown as a tree structure in Figure V.2b. Similar conventions for representing XML Schemas and DTDs have been used by previous works, e.g. [5] and [34]. Indicated are the optional (? and * labeled edges) and repeatable (* and + labeled edges) elements and the types of groups of elements (SEQ, CHOICE and ALL nodes [31].) The leaf nodes are of primitive type [13]. Like many XML Schemas, it has nesting and many “irregular” structures such as choice groups, e.g. the `body_type` may be `rectangular` or `cylindrical`, and optional elements [31], e.g. each sensor can optionally have an `image` element.

Based on the XML Schema in Figure V.2b, the Editor constructs the corresponding *EST* that serves as the basis for building the query set specification. Figure V.2c shows the Editor’s view of the *EST* as it is displayed to the developer, and Figure V.2d the internal representation used by the Editor. Formally, the *EST* is defined in the following.

Definition 14 (*Expanded Schema Tree*). *An expanded schema tree EST is a labeled tree that consists of:*

- *Element nodes n having an element name $name(n)$, which is a constant.*

Element nodes are labeled with a unique element variable $var(n)$, which starts with the \$ symbol, and an occurrence constrain $occ(n)$, which can be ? (0-1 occurrences), 1 (only one occurrences), * (any number of occurrences) or + (one or more occurrences). An element node n is optional if $occ(n)$ is either ? or *. If $occ(n)$ is either + or *, then n is repeatable. Element nodes have a Boolean property $report(n)$.

- SEQ nodes.
- CHOICE nodes.
- ALL nodes.

◇

The root node of an *EST* is a non-repeatable element node.

The Boolean property *report* of an element node is *true* if the corresponding checkbox that appears next to the element node on the view of the *EST* (Figure V.2c) is checked. The reason for doing that is to indicate to the Editor which elements to include on the report. Report generation is described in Chapter VII.

V.D.1 Aliasing and *EST* Expansion

There are cases where the developer needs to create “aliases” of element nodes. For example, assume that the developer wants to give the end-user the ability to specify two desirable protection ratings, out of the multiple that a single sensor might have. This case is depicted on Figure V.3, where two “Protection Rating” form controls appear on the query form page. To accomplish that, the developer expands the `protection_rating` element node on the *EST* and creates two copies of it, as shown on Figure V.2c. The *EST* of Figure V.2d illustrates the internal effect of the two aliases, where the two copies of the `protection_rating` element node have two different and unique element variables, `$PROT1` and `$PROT2`.

An expansion can be applied only on a repeatable element node n , creating a copy c of the subtree rooted at n and setting it as the last child of n 's parent

Next 10		Previous 10				
Image	Manufacturer	Part Number	Protection Ratings	Sensing Distance mm	Body Type	
	TURCK	BC 3-M12-AN6X	NEMA1	6.0	cylindrical	
			NEMA3		Diameter mm	Barrel Style
			NEMA4		15	Smooth
	TURCK	BC 3-M12-AP6X	NEMA3	6.0	cylindrical	
					Diameter mm	Barrel Style
	TURCK	BC 5-Q08-AN6X2	NEMA3	7.0	rectangular	
			NEMA4		Height mm	Width mm
	TURCK	BC 5-Q08-AP6X2	NEMA3	7.5	rectangular	
			NEMA6		Height mm	Width mm
	TURCK	BC 5-S18-AN4X	NEMA3	10.0	rectangular	
			NEMA11		Height mm	Width mm
	TURCK	BC 5-S18-AP4X	NEMA1	10.6	rectangular	
			NEMA3		Height mm	Width mm
	TURCK	BC 5-S18-Y0X	NEMA1	12.0	rectangular	
			NEMA3		Height mm	Width mm
	TURCK	BC 5-S185-AP4X	NEMA3	12.0	cylindrical	
					Diameter mm	Barrel Style
	TURCK	BC10-M30-AZ3X	NEMA3	15.0	cylindrical	
			NEMA13		Diameter mm	Barrel Style
	TURCK	BC10-M30-RZ3X	NEMA1	25.0	cylindrical	
			NEMA3		Diameter mm	Barrel Style
			NEMA6		20	Threaded

Figure V.3: Example *QFR* Interface

node. All element nodes of c are labeled with new and unique element variables.

V.E Example *QFR* and End-User Experience

Using QURSED, a developer can easily generate a *QFR* interface like the one shown in Figure V.3 that queries and reports proximity sensor products. This interface will be the running example and will illustrate the basic points of the functionality and the experience that QURSED delivers to the end-user of the interface.

The browser window displays a query form page and a report page. On the query form page XHTML form controls are displayed for the end-user to select

or enter desired values of sensors' attributes and customize the report page. The state of the query form page of Figure V.3 has been produced by the following end-user actions:

- Placed the equality condition “NEMA3” on “Protection Rating 1”.
- Left the preset option “No preference” on “Body Type” and placed the conditions on “Dimension X” being less than 20 “mm” and “Dimension Y” less than 40 “mm”. These two dimensions define an envelope in which the end-user wants the sensors to fit, without specifying a particular body type.
- Selected from the “Sort By Options” list to sort the results first by “Manufacturer” (descending) and then by “Sensing Distance” (ascending). The selections appear in the “Sort By Selections” list.
- In the “Customize Presentation” section, selected to present (“P” column) all columns that she has control over, e.g., “Part Number” is, by default, always presented (disabled checkbox).

After the end-user submits the form, she receives the report of Figure V.3. The results depict the information of **product** elements: the developer had decided earlier that **product** elements should be returned. By default, QURSED organizes the presentation of the qualifying XML elements in a way that corresponds to the nesting suggested by their XML Schema. Notice, for example, that each product display has nested tables for **rectangular** and **cylindrical** values. Also notice that instead of the text of the manufacturer's name, a corresponding image (logo) is presented.

Chapter VI

Tree Query Language and Query Set Specifications

This chapter describes the Tree Query Language (TQL) and the Query Set Specification (*QSS*). TQL is designed to handle the structural variance and missing fields of semistructured data, and is presented in Section VI.A. TQL's purpose is not to be yet another general-purpose semistructured query language, but to facilitate the definition of query set specifications, which are described in Section VI.B. Section VI.C illustrates how the QURSED run-time engine generates TQL queries, and in extension XQuery expressions, given a *QSS*, while Section VI.D elaborates on how the developer can apply fine control on the generation of queries using dependencies.

VI.A Tree Query Language (TQL)

End-user interaction with the query form page results in the generation of TQL queries, which are subsequently translated into XQuery statements. TQL shares many common characteristics with previously proposed XML query languages like XML-QL [23], XML-GL [22], LOREL [70], XMAS —citemix and XQuery [14]. TQL facilitates the development of query set specifications that encode large numbers of queries and the development of a visual interface for the

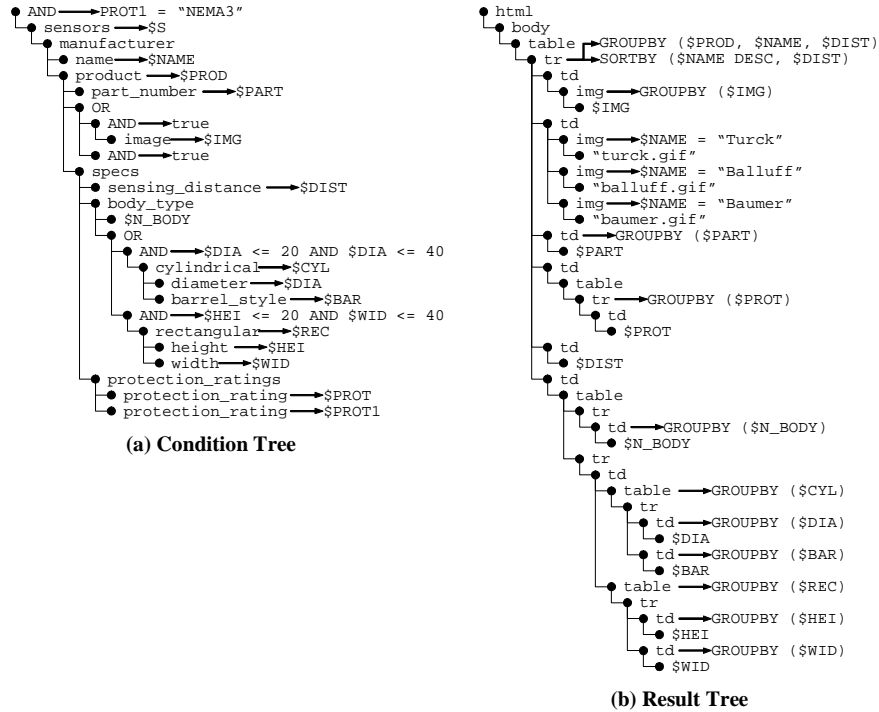


Figure VI.1: TQL Query Corresponding to Figure V.3

easy construction of those specifications. This section describes the structure and semantics of TQL queries. The structure and semantics of query set specifications are described in the next section.

A TQL query q consists of a *condition tree* and a *result tree*. An example of a TQL query is shown in Figure VI.1, and corresponds to the TQL query generated by the end-user's interaction with the query form page of Figure V.3.

VI.A.1 Condition Tree

Definition 15 (Condition Tree). *The condition tree of a TQL query q is a labeled tree that consists of:*

- *Element nodes n having an element name $\text{name}(n)$, which is a constant or a name variable, and an element variable $\text{var}(n)$. In a condition tree, there can be multiple nodes with the same constant element name, but element and name variables must be unique. Element variables start with the $\$$ symbol*

and name variables start with the $\$N_.$

- *AND nodes, which are labeled with a Boolean expression b consisting of predicates combined with the Boolean connectives \wedge , \vee and \neg . The predicates consist of arithmetic and comparison operators and functions that use element and name variables and constant values as operands and are understood by the underlying query processor. Each element and name variable used in b belongs to at least one element node that is either an ancestor of the AND node, or a descendant of the AND node such that the path from the AND node to the element node does not contain any OR nodes. The Boolean expression may also take the values true and false.*
- *OR nodes.* ◇

The following constraints apply to condition trees:

1. The root element node of a condition tree is an AND node.
2. OR nodes have AND nodes as children.

Figure VI.1 shows the TQL query for the example of Figure V.3. Note that two conditions are placed on diameter of cylindrical sensors corresponding to height and width of rectangular sensors. Omitted are the variables that are not used in the condition or the result tree.

The semantics of condition trees is defined in two steps: OR-removal and binding generation. OR-removal is the process of transforming a condition tree with OR nodes into a forest of condition trees without OR nodes, called *conjunctive condition trees* in the remainder of the chapter. OR-removal for the condition tree of Figure VI.1a results in the set of the four condition trees shown in Figure VI.2.

Intuitively, OR-removal is analogous to turning a logical expression to disjunctive normal form [39]. In particular, we repeatedly apply the rules shown in Figure VI.3. Without loss of generality, the subtrees of Figure VI.3 are presented

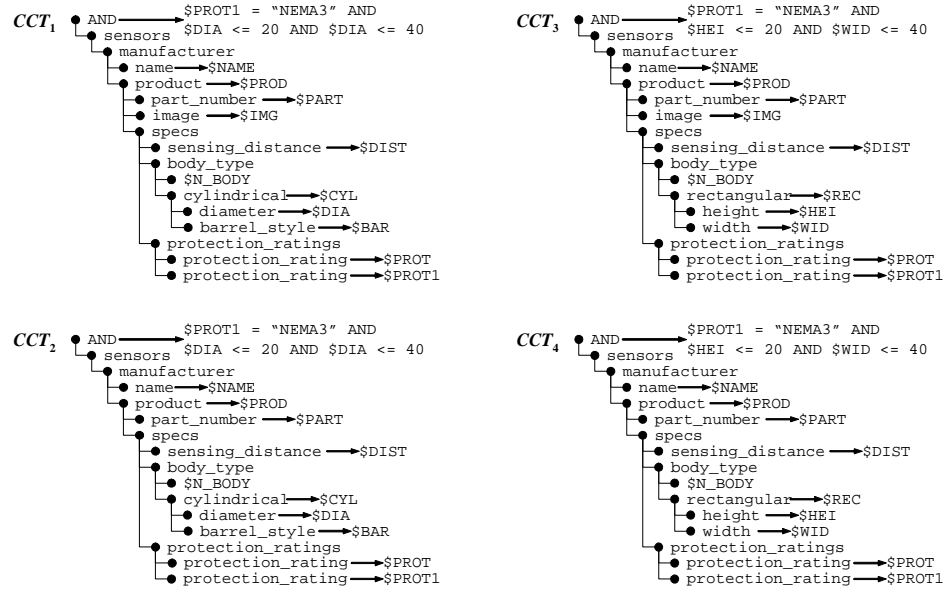


Figure VI.2: Conjunctive Condition Trees

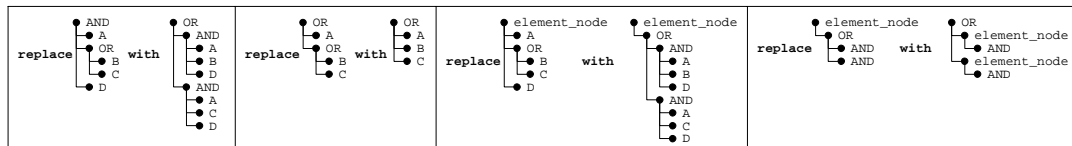


Figure VI.3: OR-Removal Replacement Rules

with 2 or 3 children. At the point when we cannot apply the rules further, we have produced a tree with an OR root node, which we replace with the forest of conjunctive condition trees consisting of all the children of the root OR node. Notice that wherever this process generates AND nodes as children of AND nodes, these can be merged, and the Boolean expression of the merged node is the conjunction of the Boolean expressions of the original AND nodes. Also notice that the Boolean expression of the root AND node in the first rule cannot contain any variables in subtrees B or C, per earlier definition of condition trees. Finally, notice that in the course of OR-removal “intermediate results” may not be valid condition trees per Definition 15 (in particular, constraint 2 can be violated), but the final results obviously are. The semantics of the original condition tree is given in terms of the semantics of the resulting conjunctive condition trees.

A conjunctive condition tree C produces all bindings for which an input *loto* t “satisfies” C . Formally, a binding is a mapping β from the set of element variables and name variables in C to the nodes and node labels of t , such that the child of the root of C (which is an AND node) matches with the root of t , i.e., $\beta(\text{var}(\text{child}(\text{root}(C)))) = \text{root}(t)$, and recursively, traversing the two trees top-down, for each child n_i of an element node n in C , assuming $\text{var}(n)$ is mapped to a node x in t , there exists a child x_i of x , such that $\beta(\text{var}(n_i)) = x_i$ and, if x_i is not a leaf node:

- if $\text{name}(n_i)$ is a constant, $\text{name}(n_i) = \text{name}(x_i)$
- if $\text{name}(n_i)$ is a name variable, $\beta(\text{name}(n_i)) = \text{name}(x_i)$

Importantly, AND nodes in C are ignored in the traversal of C . In particular, in the definition above, by “child of the element”, we mean either element child of the element, or the child of an AND node that is the child of the element. A binding is *qualified* if it makes *true* the Boolean expressions that label the AND nodes of C . Notice that it is easy to do AND-removal on conjunctive condition trees. Let a_1, \dots, a_n be the AND nodes in a *CCT* with root a , and let b_1, \dots, b_n , and b be their Boolean expressions. We can eliminate a_1, \dots, a_n , and replace b with b AND b_1 and \dots and b_n .

The result of C is the set of qualified bindings. For a conjunctive condition tree with element and name variables $\$V_1, \dots, \V_k , a binding is represented as a tuple $[\$V_1 : v_1, \dots, \$V_k : v_k]$ that binds $\$V_i$ to node or value v_i , where $1 \leq i \leq k$. A binding of some of the variables in a (conjunctive) condition tree is called a *partial* binding. Note that the semantics of a binding requires total tuple assignment [70], i.e., every variable binds to a node or a string value.

The semantics of a condition tree is defined as the union of the bindings returned from each of the conjunctive condition trees in which it is transformed by OR-removal. For example, the result of the four conjunctive condition trees shown in Figure VI.2 on the source *loto* of Figure V.2a is shown in Table VI.1. The union

\$NAME	\$PROD	\$PART	\$IMG	\$DIST	\$N_BODY	\$CYL	\$DIA	\$BAR	\$PROT	\$PROT1				
Turck	product part_number "A123"	A123	A123.jpg	11	cylindrical	cylindrical diameter "17"	17	Smooth	NEMA1	NEMA3	CCT_1			
Turck	product part_number "A123"	A123	A123.jpg	11	cylindrical	cylindrical diameter "17"	17	Smooth	NEMA3	NEMA3	CCT_1			
Turck	product part_number "A123"	A123		11	cylindrical	cylindrical diameter "17"	17	Smooth	NEMA1	NEMA3	CCT_2			
Turck	product part_number "A123"	A123		11	cylindrical	cylindrical diameter "17"	17	Smooth	NEMA3	NEMA3	CCT_2			
Turck	product part_number "B123"	B123		25	rectangular				rectangular height "10"	10	30	NEMA3	NEMA3	CCT_4
Turck	product part_number "B123"	B123		25	rectangular				rectangular height "10"	10	30	NEMA4	NEMA3	CCT_4

Table VI.1: Bindings for Conjunctive Condition Trees of Figure VI.2

of the sets of bindings does not need to remove duplicate bindings or bindings that are subsumed by other bindings (e.g., CCT_2 rows are subsumed by CCT_1 rows in Table VI.1.) The necessary duplicate elimination is performed during construction. Notice that three of the four conjunctive condition trees generate two bindings each. Notice also that the union is heterogeneous, in the sense that the conjunctive condition trees can contain different element variables and thus their evaluation produces heterogeneous binding tuples.

The above shows that the semantics of an OR node is that of union and it cannot be simulated by a disjunctive Boolean condition labeling an AND node. OR nodes therefore are necessary for queries over semistructured data sources (e.g., sources whose XML Schema makes use of choice groups and optional elements.)

The condition tree corresponds intuitively to the WHERE part of XML query languages such as XML-QL, LOREL and XMAS, to the *extract* and *match* parts of XML-GL, and to the FOR and WHERE clauses of a FLWOR expression of XQuery. The result tree correspondingly maps to the CONSTRUCT clause of XML-QL and XMAS, the SELECT clause of LOREL, the *clip* and *construct* parts of XML-GL, and the RETURN clause of a FLWOR expression of XQuery. A result tree specifies how to build new XML elements using the bindings provided by the condition tree.

VI.A.2 Result Tree

Definition 16 (Result Tree). *A result tree of a TQL query q is a node-labeled tree that consists of:*

- *Element nodes n having an element name $\text{name}(n)$, which is a constant if n is an internal node, and a constant or a variable that appears in the condition tree of q , if n is a leaf node.*
- *A group-by label G and a sort-by label S on each node. A group-by label G is a (possibly empty) list of variables $[\$V_1, \dots, \$V_n]$ from the condition tree of q . A sort-by label S is a list of $(\$V_i, O_i)$ pairs, where $\$V_i$ is a variable from the condition tree of q , and O_i is the sorting order determined for $\$V_i$. O_i can take the values “DESC” for descending or “ASC” for ascending order. Each variable in the sort-by list of a node must appear in the group-by list of the same node. Empty group-by and sort-by labels are omitted from figures in the remainder of the chapter.*
- *A Boolean expression b on each node consisting of predicates combined with the Boolean connectives \wedge , \vee and \neg . The predicates consist of arithmetic and comparison operators and functions that use element and name variables appearing in the condition tree of q , and constant values as operands. \diamond*

Every element or name variable must be in the scope of some group-by list or Boolean condition. Similar to logical quantification, the scope of a group-by list or a Boolean condition of a node is the subtree rooted at that node. Figure VI.1b shows the result tree for the example of Figure V.3. Note that the rows of the XHTML tables that contain the static column names are omitted from the result tree for presentation clarity. Group-by and sort-by labels are the TQL means of performing grouping and sorting. The intuition behind Boolean expressions on nodes is that they provide control on the construction of nodes in the result of a query: A node (and its subtree) is only added to the result of the query if there is

at least one qualified binding of the variables in the condition for that node that renders it *true*.

Given a TQL query with condition tree and result tree, the answer of the query on given input is constructed from the set of qualified bindings of the condition tree. In what follows, binding refers to qualified binding. The result is a *loto* constructed by structural recursion on the result tree as formally described below. The recursion uses partial bindings to instantiate the group-by variables and condition variables of element nodes.

Traversing the result tree top-down, for each subtree $tree(n)$ rooted at element node n with group-by label $[\$V_1, \dots, \$V_k]$ and, without loss of generality, sort-by label $[\$V_1, \dots, \$V_m]$ ($m \leq k$), let $\mu = [\$V_{A1} : v_{A1}, \dots, \$V_{An} : v_{An}]$ be a partial binding that instantiates all the group-by and condition variables of the ancestors of n , let the Boolean expressions of n and its ancestors be b and b_{A1}, \dots, b_{Ah} , and let the variables in these expressions that do not appear among the $[\$V_{A1}, \dots, \$V_{An}, \$V_1, \dots, \$V_k]$ be $[\$B_1, \dots, \$B_j]$. Recursively replace the subtree $tree(n)$ in place with a list of subtrees, one for each qualified binding $\pi = [\$V_{A1} : v_{A1}, \dots, \$V_{An} : v_{An}, \$V_1 : v_1, \dots, \$V_k : v_k]$ such that v_1, \dots, v_m are string values, by instantiating all occurrences of $\$V_{A1}, \dots, \$V_{An}, \$V_1, \dots, \V_k with $v_{A1}, \dots, v_{An}, v_1, \dots, v_k$, if and only if b, b_{A1}, \dots, b_{Ah} all evaluate to true for some qualified binding $\pi' = [\$V_{A1} : v_{A1}, \dots, \$V_{An} : v_{An}, \$V_1 : v_1, \dots, \$V_k : v_k, \$B_1 : b_1, \dots, \$B_j : b_j]$ (otherwise the subtree is not included in the list of subtrees produced.) The list of instantiated subtrees is ordered according to the conditions in the sort-by label.

Figure VI.4 shows the resulting *loto* from the TQL query of Figure VI.1 and the bindings of Table VI.1. Note, for example, that for each of the two distinct partial bindings of the triple $[\$PROD, \$NAME, \$DIST]$, one **tr** element node is created, and that, for each such binding, different subtrees rooted at the nested **table** element nodes are created, corresponding to different π bindings. Finally, out of the three Boolean expressions that label the **img** elements in Figure VI.1b,

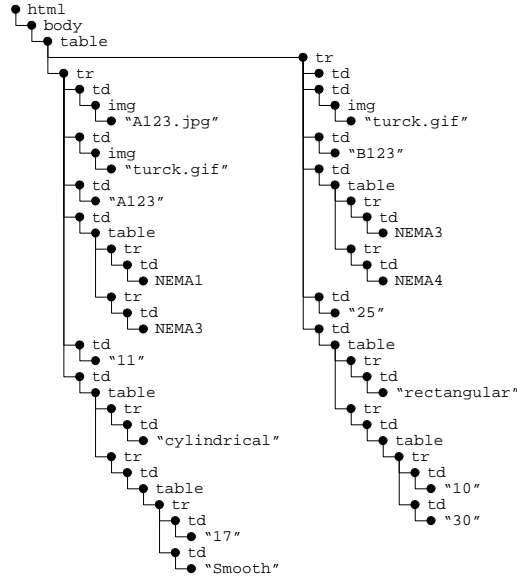


Figure VI.4: Resulting *loto* for Bindings of Table VI.1

only the first one evaluates to *true*, for both sensors, based on the bindings of variable `$NAME` in Table VI.1.

The QURSED system uses the TQL queries internally, but issues queries in the standard XQuery language by translating TQL queries to equivalent XQuery statements. The algorithm for translating TQL queries to equivalent XQuery statements is given in Appendix F. The XQuery specification is a working draft of the World Wide Web Consortium (W3C); for a more detailed presentation of the language and its semantics see [14] and [25].

The TQL query generated by a query form page is a member of the set of queries encoded in the query set specification of the *QFR*. The next section describes the syntax and semantics of query set specifications.

VI.B Query Set Specification

Query set specifications are used by QURSED to succinctly encode in *QFRs* large numbers of possible queries. In general, the query set specification can describe a number of queries that is exponential in the size of the specification. The

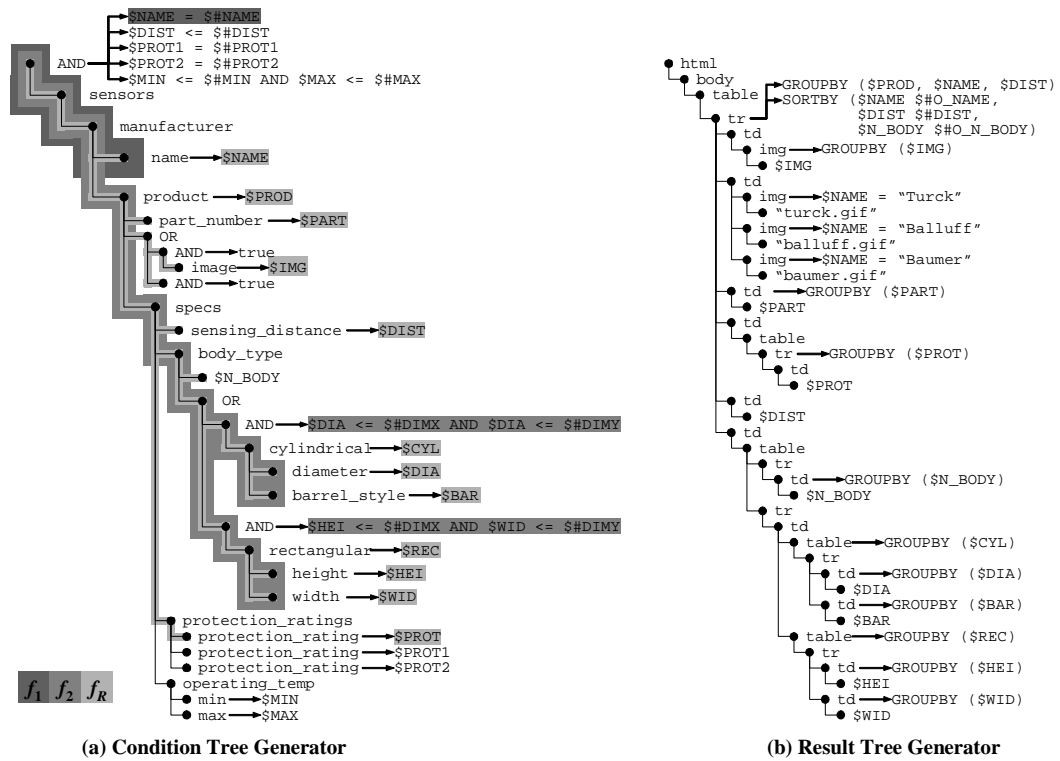


Figure VI.5: Query Set Specification

specification also includes a set of dependencies that constrain the set of queries that can be produced.

The developer uses the Editor to visually create a query set specification, like the one in Figure VI.5. This section formally presents the query set specification that is the logical underpinning of *QFRs*.

Definition 17 (Query Set Specification). *A query set specification QSS is a 4-tuple $\langle \text{CTG}, \text{RTG}, F, D \rangle$, where:*

- CTG, the condition tree generator, is a condition tree with three modifications:
 - AND nodes a_i can be labeled with a set of Boolean expressions $B(a_i)$.
 - The same element or name variable can appear in more than one condition fragments.
 - Boolean expressions can use parameters (a.k.a. placeholders [50]) as operands of their predicates. Parameters are denoted by the $\$ \#$ symbol and must bind to a value [13].

The same constraints apply to a CTG as to a condition tree.

- RTG, the result tree generator, is a result tree with two modifications. First, the variables that appear in the sort-by label S on a node do not have a specified order (ascending or descending,) as in the case of a result tree, but they have a parameter instead, called ordering parameter that starts with the $\$ \# O_$. Second, the Boolean expressions on nodes can use parameters as operands of their predicates. Boolean expressions on nodes involving only parameters and constants as operands (no variables) are a special case since they can be evaluated as soon as the parameters are instantiated. Their use is described later in Section VII.F of Chapter VII.
- F is a non-empty set of condition fragments. A condition fragment f is defined as a subtree of the CTG, rooted at the root node of the CTG, where

each AND node a_i is labeled with exactly one Boolean expression $b \in B(a_i)$. Each variable used in b must belong to a node included in f . F always contains a special condition fragment f_R , called result fragment, that includes all the element nodes whose variables appear in the RTG, all its AND nodes are labeled with the Boolean value true, and has no parameters. The result fragment intuitively guarantees the “safety” of the result tree.

- D is an optional set of dependencies. Dependencies are defined in Section VI.D. ◇

For example, the query set specification of Figure VI.5 encodes, among others, the TQL query of Figure VI.1. The CTG in Figure VI.5a corresponds partially to the set F of condition fragments defined for the query form page of Figure V.3. Three condition fragments are indicated with different shades of gray:

1. condition fragment f_1 is defined by the dark grey subtree and the Boolean expression on the root AND node of the CTG that applies a condition to the name element node;
2. condition fragment f_2 is defined by the medium gray subtree and the Boolean expressions that apply a condition to the dimensions of cylindrical and rectangular sensors ; and
3. condition fragment f_R (the *result fragment*) is defined by the light grey subtree that includes all the element nodes whose variables appear in the RTG in Figure VI.5b, and imposes no Boolean conditions.

How the developer produces a query set specification via the Editor is described in Chapter VII.

VI.C Query Formulation Process

Figure VI.6 summarizes the query formulation process of the QURSED run-time engine. The process starts by accepting a $QSS(CTG, RTG, F, D)$ and

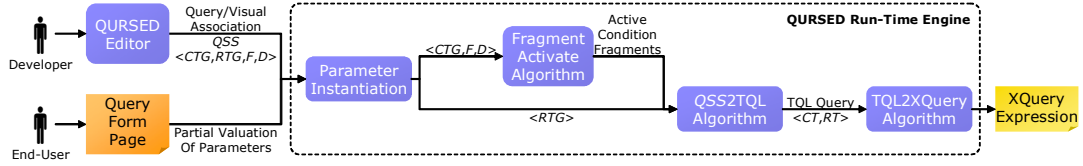


Figure VI.6: Query Formulation Process

a query/visual association, provided by the interaction of the developer with the Editor, and a partial valuation of its parameters, provided by the end-user’s interaction with the query form page. The process terminates by outputting an XQuery expression.

Parameter Instantiation. The run-time engine first instantiates the parameters of the condition tree generator CTG and the result tree generator RTG . In particular, during the end-user’s interaction with the query form page, and based on which form controls she fills out and on the query/visual association, a partial valuation v over P , where P is the set of the parameters that appear in the QSS , is generated. As an example partial valuation, consider the one generated by the query form page of Figure V.3 from the constant values the end-user provides:

$$v = \{\$ \# \text{PROT1:} \text{“NEMA3”}, \$ \# \text{DIMX:} \text{“20”}, \$ \# \text{DIMY:} \text{“40”}, \$ \# \text{O_NAME:} \text{“DESC”}, \$ \# \text{O_DIST:} \text{“ASC”}\}$$

Based on v , the run-time engine instantiates the parameters of condition fragments in F . For example, the above partial valuation instantiates the parameters $\$ \# \text{DIMX}$ and $\$ \# \text{DIMY}$ of condition fragment f_2 of Figure VI.5a, which imposes a condition on the dimensions of the sensor’s body type. Similarly, the ordering parameters of the sort-by labels of the RTG , and the parameters of Boolean expressions labeling nodes of the RTG , are instantiated. The ordering parameters can take the values “DESC” or “ASC”, as in the case of $\$ \# \text{O_NAME}$ and $\$ \# \text{O_DIST}$ in the above partial valuation. An example of an RTG , where parameterized Boolean expressions are labeling its nodes, is shown in Section VII.F of Chapter VII. Finally, the run-time engine also instantiates the parameters of the set of dependencies D . Dependencies are presented in the next section.

FragmentActivate Algorithm. As a second step on Figure VI.6, the `FragmentActivate` algorithm inputs the instantiated *CTG* and the set of condition fragments F , and outputs the set of active condition fragments. The algorithm renders a condition fragment *active* if it has all its parameters instantiated by the partial valuation v . Since the partial valuation v might not provide values for all the parameters used in the *CTG*, some condition fragments are rendered *inactive*. Based on the above example partial valuation, condition fragment f_2 of Figure VI.5a and the condition fragment that imposes a condition on protection rating (not indicated in Figure VI.5a) are rendered active, while condition fragment f_1 on manufacturer's name is inactive, since parameter `$#NAME` is not instantiated by v . As a special case, the result fragment f_R is always active, since it does not have any parameters.

Note that the `FragmentActivate` algorithm on Figure VI.6 also inputs the set of dependencies D , which further complicate the algorithm. Both the dependencies and the revised version of the `FragmentActivate` algorithm are presented in the next section.

QSS2TQL Algorithm. The set of active condition fragments and the instantiated *RTG* are passed to the *QSS2TQL* algorithm, which outputs a TQL query by formulating its condition tree CT and its result tree RT . The CT consists of the union of the nodes of the active condition fragments f_1, \dots, f_n , along with the edges that connect them. Each AND node n_{AND} in the CT is annotated with the conjunction $c_1 \wedge \dots \wedge c_n$ of the Boolean expressions c_1, \dots, c_n that annotate the node n_{AND} in the fragments f_1, \dots, f_n respectively.

Similarly, in order to convert the *RTG* to the RT , the *QSS2TQL* algorithm first eliminates from the *RTG* the subtrees rooted at nodes labeled with a Boolean expression b that has uninstantiated parameters or evaluates to *false*, as further explained in Section VII.F of Chapter VII. Then for every node that has a sort-by label S , we keep in the label only the variables with instantiated ordering parameters.

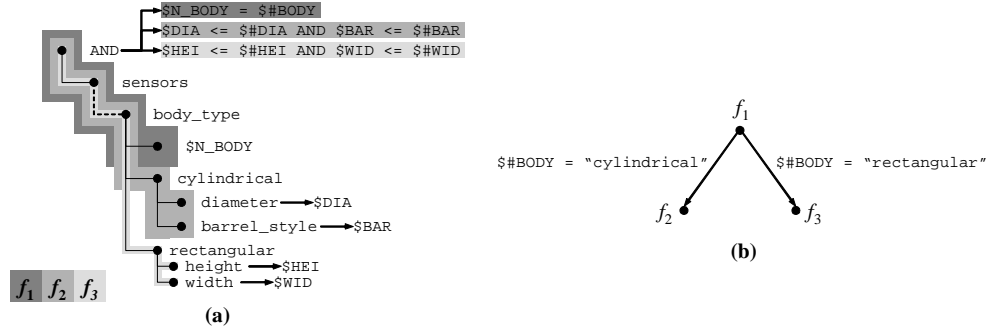


Figure VI.7: Condition Tree Generator and Dependencies Graph

As an example of the *QSS2TQL* algorithm, consider the *CT* of Figure VI.1a, which is formulated based on the active condition fragments of Figure VI.5a, i.e., f_2 , the condition fragment that imposes a condition on protection rating, and the result fragment f_R . Accordingly, the *RT* of Figure VI.1b is formulated from the *RTG* of Figure VI.5b, where the variable $\$N_BODY$ is excluded from the top sort-by list, since its ordering parameter $\$O_N_BODY$ is not instantiated by the example partial valuation above.

TQL2XQuery Algorithm. The final step of the query formulation process on Figure VI.6 passes the TQL query as input to the TQL2XQuery algorithm, presented in Appendix F. The TQL2XQuery algorithm outputs the final XQuery expression, which is sent to the underlying XQuery processor.

VI.D Dependencies

Dependencies allow the developer to define conditions that include or exclude condition fragments from the condition tree depending on the end-user's input. Dependencies provide a flexible way to handle data irregularities and structural variance in the input data, and a declarative way to control the appearance of visual fragments.

Definition 18 (*Dependency*). A dependency d is defined as a 3-tuple $\langle f, B, H \rangle$ over a set of condition fragments F , where $f \in F$ is the dependent condition frag-

Mechanical	
Body Type	Cylindrical
Diameter	<input type="text"/> mm
Barrel Style	No preference

(a)

Mechanical	
Body Type	Rectangular
Height	<input type="text"/> mm
Width	<input type="text"/> mm

(b)

Figure VI.8: Dependencies on the Query Form Page

ment and B is the condition of the dependency consisting of predicates combined with the Boolean connectives \wedge , \vee and \neg . The predicates consist of arithmetic and comparison operators and functions that use parameters from the CTG and constant values as operands. The set $H \subseteq F$, called the head of the dependency, contains the condition fragments that use at least one parameter that appears in B . ◇

A dependency d holds if each parameter p_i in B is instantiated in a condition fragment in H that is active, and B evaluates to *true*. In the presence of dependencies, a fragment f is active if all its parameters are instantiated *and* at least one of the dependencies, where f is the dependent condition fragment, holds. Intuitively, a set of dependencies constrains the set of queries a query set specification can generate by rendering inactive the dependent condition fragments when none of their dependencies hold. For example, consider the condition tree generator and condition fragments of Figure VI.7a, and let us define two dependencies d_1 and d_2 as follows:

$$\langle f_2, \text{\$#BODY} = \text{"cylindrical"}, \{f_1\} \rangle \quad (d_1)$$

$$\langle f_3, \text{\$#BODY} = \text{"rectangular"}, \{f_1\} \rangle \quad (d_2)$$

The condition fragment f_1 uses the parameter $\text{\$#BODY}$ that appears in the condition of both dependencies on f_2 and f_3 . If a value is not provided for $\text{\$#BODY}$, then neither dependency holds, and f_2 and f_3 are inactive. If the value "cylindrical" is provided, then f_1 is active, the condition for d_1 is *true*, and so f_2 is rendered active.

Dependencies affect the appearance of a query form. More specifically, QURSED hides from the query form page those visual fragments whose condition

fragments participate in dependencies that do not hold. For example, Figure VI.8 demonstrates the effect of dependencies d_1 and d_2 on the query form page of Figure V.3. The two shown sets of form controls are the visual fragments of the condition fragments shown in Figure VI.7a. For instance, the condition fragment f_1 applies a condition to the element node labeled with $\$BODY$ and its visual fragment consists of the “Body Type” form control. End-user selection of the “Cylindrical” option in the “Body Type” form control results in having d_1 hold, which makes the visual fragment for f_2 visible (Figure VI.8a.) Notice that f_2 is still inactive: values for “Diameter” and “Barrel Style” need to be provided. Notice also that an inactive condition fragment whose dependencies do not hold has no chance of becoming active in QURSED: its visual fragment is hidden, so there is no way for the end-user to provide values for the parameters of the condition fragment.

Obviously, circular dependencies must be avoided, since the involved dependent fragments can never become active. This restriction is captured by the *dependency graph*:

Definition 19 (*Dependency Graph*). A *dependency graph* for a set of dependencies D and a set of condition fragments F is a directed labeled graph $G = \langle V, E \rangle$, where the nodes V are the condition fragments in F and for every dependency d in D there is an edge in E from every condition fragment f_i in the head H of d to the dependent condition fragment f , labeled with the condition B of d . \diamond

The dependency graph for the dependencies d_1 and d_2 defined above is shown in Figure VI.7b. QURSED enforces that the dependency graph is *acyclic*.

The QURSED system activates the appropriate visual fragments (updating the query form page) and condition fragments, based on which parameters have been provided and which dependencies hold. The algorithm for “resolving” the dependencies to decide which fragments are active, called *FragmentActivate*, is based on topological sort [47] (hence of complexity $\Theta(V + E)$) and is outlined below. Note that, when evaluating a condition b of a dependency, any predicates that contain uninstantiated parameters evaluate to *false*.

Algorithm FragmentActivate

Inputs: A dependencies graph $G = \langle V, E \rangle$, and a partial valuation v over P , where P is the set of the parameters that appear in the QSS .

Output: The set A of active condition fragments.

Method:

$A \leftarrow \emptyset$	1
Compute the set of fragments B , whose parameters are all instantiated by v	2
For each edge (n, u) in E	3
Evaluate the condition on edge (n, u)	4
Repeat	5
If node u belongs to B and has no incoming edges	6
$A \leftarrow \{u\}$	7
If node u belongs to B , has an incoming edge (n, u) ,	8
where n belongs in A and the condition on (n, u) is true	
$A \leftarrow \{u\}$	9
Until A reaches fixpoint	10

Section VII.C of Chapter VII describes how the developer can define dependencies using the Editor.

Chapter VII

Editing Query Set Specifications

This chapter presents the QURSED Editor that is the visual tool for the development of web-based query forms and reports and their Query Set Specification (*QSS*). Section VII.A describes the overall architecture of the Editor. Section VII.B presents the visual actions, algorithms and heuristics employed by the Editor for building the Condition Tree Generator (*CTG*) of a *QSS*. Section VII.C discusses the visual process for building dependencies among condition fragments. Section VII.D elaborates on the generation of the Report Tree Generator (*RTG*) of a *QSS*, which can either be schema-driven or based on a template report page. Section VII.E and Section VII.F present the customization functionality that QURSED provides on report pages.

VII.A Architecture

The QURSED Editor is the tool the developer uses to build *QFRs*. Figure VII.1 shows the Editor's architecture, how the developer interacts with the graphical user interface, and how the Editor interprets these visual actions in order to construct the *QSS* and the query/visual association of a *QFR*.

The developer builds a condition tree generator by constructing a set of Boolean expressions based on the input XML Schema, in the form of an *EST*, and the input XHTML query form page that are displayed to her. Internally, the

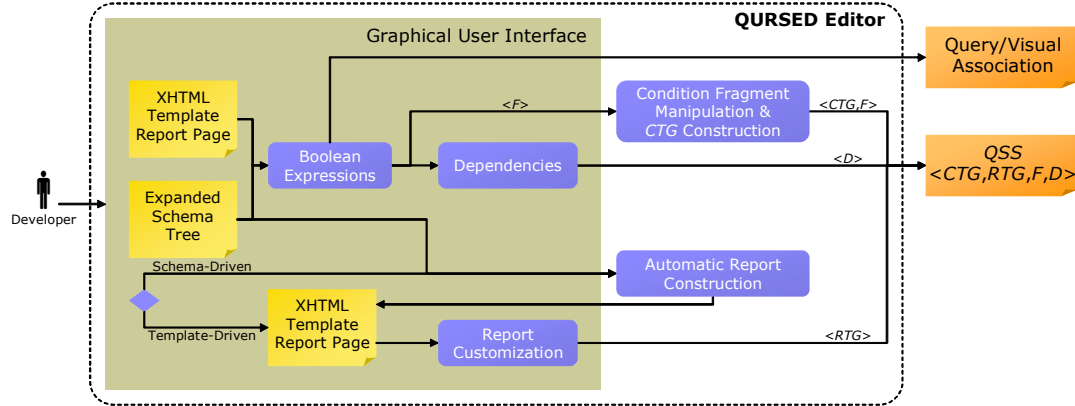


Figure VII.1: QURSED Editor Architecture

Editor interprets the set of Boolean expressions as the set of condition fragments of the *QSS* and the query/visual association. The Editor constructs the *CTG* by building each condition fragment f , as if f was the only fragment of the condition tree generator, and then merging f with the *CTG*. A key step in that process is that the Editor checks if f is meaningful by considering the presence of CHOICE elements in the *EST* and, if necessary, manipulates f by introducing heuristically structural disjunction operators (OR nodes). The developer also builds the set of dependencies on the set of condition fragments that become part of the *QSS*. These processes are described in Section VII.B and Section VII.C.

For the construction of the result tree generator, the developer has two choices that are illustrated as a diamond on Figure VII.1. Either an XTMHL template report page is automatically constructed based on the *EST* (schema-driven), or one is provided as an input (template-driven). Either way, the Editor constructs internally an *RTG* that becomes part of the *QSS*. This process is described in Section VII.D. The developer can also further customize the template report page report by building Boolean expressions and adding dynamic projection functionality, presented in Section VII.E and VII.F.

A key benefit of the Editor is that it enables the easy generation of semi-structured queries with OR nodes by considering the presence of CHOICE elements

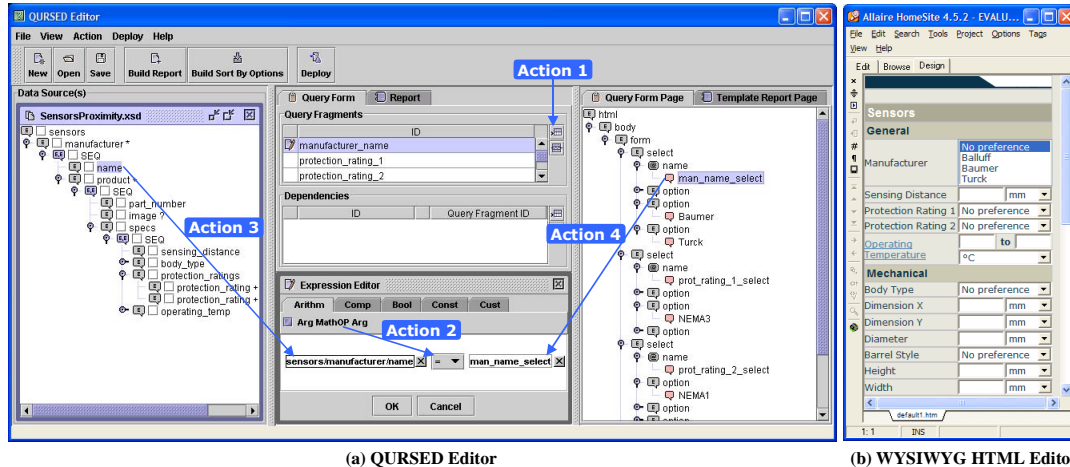


Figure VII.2: Building a Condition Fragment

in the *EST*. The following sections describe the visual actions and their translation to corresponding parts of the query set specification, using the *QSS* of Figure VI.5 and the *QFR* of Figure V.3 as an example.

VII.B Building Condition Tree Generators

Figure VII.2a demonstrates how the developer uses the Editor to define the condition fragment f_1 of Figure VI.5a. The main window of the Editor presents the sample *EST* of Section V.D on the left panel, and the query form page on the right panel. The query form page is displayed as an XHTML tree that contains a form element node and a set of form controls, i.e., `select` and `input` element nodes [71]. The XHTML tree corresponds to the page shown on Figure VII.2b rendered in the [43] WYSIWYG XHTML editor. Based on this setting, the developer defines the condition fragment f_1 of Figure VI.5a that imposes an equality condition on the manufacturer’s name by performing the four actions indicated by the arrows on Figure VII.2a.

The developer starts by clicking on the “New Condition Fragment” button (Action 1 of Figure VII.2a) and providing a unique ID, which is `manufacturer_name` in this case. The middle panel lists the condition fragments defined so far, and

the expression editor at the bottom allows their definition, inspection and revision. Then, the developer builds a Boolean expression in the expression editor, by drag and dropping the equality predicate (Action 2) and setting its left operand to be the element node name (Action 3). The full path name of the node appears in the left operand box and is also indicated by the highlighting of the name element node on the left panel. As a final step, the developer binds the right operand of the equality predicate to the select XHTML form control named `man_name_select` (Action 4) thus establishing a query/visual association and defining as the visual fragment the “Manufacturer” form control shown in Figure VII.2b. Internally, the Editor creates the parameter `$#NAME`, associated with the “Manufacturer” form control of Figure VII.2b, and sets it as the right operand of the Boolean expression, as Figure VI.5a shows.

In order to build more complex condition fragments, Actions 2, 3 and 4 can be repeated multiple times, thus introducing multiple variable and parameters and including more than one XHTML form controls in the corresponding visual fragment.

Note that, even though the visual actions introduce variables and parameters in the condition fragment, the developer does not need to be aware of their names. In effect, variables correspond to path names and parameters to XHTML form control names. The Editor interprets the Boolean expression as a condition fragment that contains all paths of the expression.

VII.B.1 Automatic Introduction of Structural Disjunction

The semistructuredness of the schema (CHOICE nodes and optional elements) may render the Boolean expression meaningless and unsatisfiable. The Editor automatically, and by employing a heuristic, manipulates a condition fragment f by introducing structural disjunction operators (OR nodes) that render f meaningful.

For example, consider the query form page of Figure VII.2b, where the

end-user has the option to input two dimensions X and Y that define an envelope for the sensors, without specifying a particular body type. Sensors can be either cylindrical or rectangular. The developer’s intention is to specify that either the diameter is less than dimensions X and Y , or the height is less than dimension X and the width less than Y . The developer constructs the following Boolean expression by following the previously described steps:

$$(\$DIA \leq \$\#DIMX \wedge \$DIA \leq \$\#DIMY) \vee (\$HEI \leq \$\#DIMX \wedge \$WID \leq \$\#DIMY)$$

The $\$DIA$, $\$HEI$ and $\$WID$ variables label the diameter, height and width elements of the *EST*. The $\#\#DIMX$ and $\#\#DIMY$ parameters are associated with the “Dimension X” and “Dimension Y” form controls.

However, the query where the above Boolean expression is interpreted as a condition fragment consisting of the paths to `diameter`, `height` and `width` elements is unsatisfiable, since no sensor has all of them. The Editor captures the original intention by automatically manipulating the \vee Boolean connective and treating it as an OR node of TQL, as the condition fragment f_2 in Figure VI.5a indicates. The OR node corresponds to the CHOICE node in the *EST* of Figure V.2c. Two AND nodes are also introduced and are labeled with the conjunctions in the initial Boolean expression: $(\$DIA \leq \$\#DIMX \wedge \$DIA \leq \$\#DIMY)$ and $(\$HEI \leq \$\#DIMX \wedge \$WID \leq \$\#DIMY)$. The manipulation of a condition fragment is part of the Construct *CTG* algorithm.

The Construct *CTG* algorithm creates a condition tree generator by merging the condition fragments. It operates incrementally by merging each condition fragment f with the condition tree generator already constructed from the previous condition fragments. The main step of the algorithm manipulates f by employing a heuristic, such that f produces meaningful satisfiable queries given the Boolean expression b . In particular, the algorithm introduces structural disjunction operators to f by replacing Boolean connectives \vee in b with OR nodes, as illustrated in the example above. The manipulation is driven by the CHOICE nodes and

optional elements. An initial step of the algorithm checks if f can be manipulated to produce meaningful, satisfiable queries. This is accomplished by bringing b to disjunctive normal form and identifying at least one unsatisfiable conjunction. If there is one, then the algorithm terminates outputting an error. The final step of Construct CTG merges f with the input CTG . The order that the condition fragments are passed to the algorithm does not matter.

The Construct CTG algorithm assumes a function $node(\$V_i)$ that given a variable name $\$V_i$ in b returns the node n_i of the EST that the variable corresponds to, i.e., the node of the EST that the developer drag and dropped. In the case of name variables, $node(\$V_i)$ returns the parent of the node that the developer drag and dropped. It also assumes the existence of a function $copy(n_i)$ that, given a node n_i in the EST , returns the copy of it in f , if there exists one, or *null*, otherwise.

Algorithm Construct CTG

Inputs: A condition fragment f with a Boolean expression b labeling its root AND node, a condition tree generator CTG , and an EST .

Output: The condition tree generator CTG where f has been added, or an error if f cannot produce satisfiable queries.

Method:

Step 1: Satisfiability Check of f

Rewrite b in disjunctive normal form such that $b = c_1 \vee c_2 \dots \vee c_n$, 1

where c_i is a conjunction of predicates

If a conjunction c_i , where $1 \leq i \leq n$, uses two variables $\$V_{ix}$, $\$V_{iy}$ 2

such that the lowest common ancestor of $node(\$V_{ix})$ and $node(\$V_{iy})$ in the EST is a CHOICE node

Output an error indicating the unsatisfiable conjunctions 3

Step 2: Manipulation of f

// Introduces OR nodes to f based on CHOICE nodes in the EST

For any two variables $\$V_{ix}$, $\$V_{jy}$ used in conjunctions c_i and c_j of b , 4

respectively, where $1 \leq i, j \leq n$ and $i \neq j$

If both the paths from $node(\$V_{ix})$ and $node(\$V_{jy})$ to their lowest element 5

node common ancestor n_{ANSC} in the EST contain either a CHOICE node or an optional element, excluding n_{ANSC}

Apply the Rules 1 and 2 of Figure VII.3 6

// Label AND nodes with Boolean expressions

For each conjunction c_i of b , $1 \leq i \leq n$ 7

In f , identify the lowest AND node a_i that is the common ancestor of all 8

the element nodes labeled with the variables used in c_i and label it with Boolean expression c_i

If the AND node is labeled with more than one conjunctions 9

Combine them with the \vee Boolean connective 10

Step 3: Addition of f to the CTG

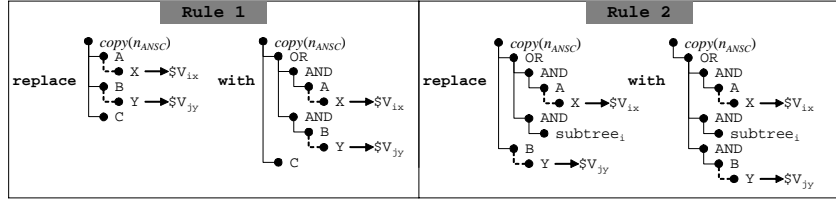


Figure VII.3: “OR Node Introduction” Rules

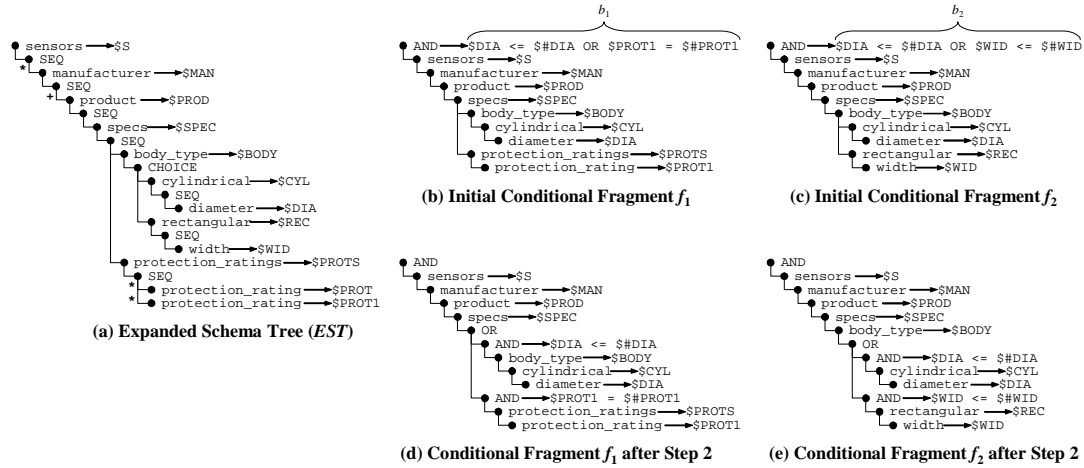


Figure VII.4: Example of the Construct *CTG* Algorithm

- Set the children of the root AND node of f as children of the root AND node of the CTG 11
- Take the union of the sets of Boolean expressions labeling the root AND node of f and the root AND node of the CTG and label the root AND node of the latter with it 12

Line 6 of the algorithm covers two cases that are illustrated in Figure VII.3. In the first case, the node $copy(n_{ANSC})$ does not have an OR child node and Rule 1 shows how the condition fragment f is manipulated. In the second case the node $copy(n_{ANSC})$ has an OR child node n_{OR} and the subtree $tree_{ix}$ that contains $node(\$V_{ix})$ is a child of an AND child node n_{AND} of n_{OR} , and $tree_{jy}$ that contains $node(\$V_{jy})$ is a child of $copy(n_{ANSC})$. In this case, Rule 2 does not introduce a new OR node, but places the subtree rooted at B under the existing OR node instead.

Figure VII.4 illustrates an example application of the Construct *CTG* al-

gorithm on the condition fragments defined on the *EST* of Figure VII.4a. Assume the developer has built two Boolean expressions b_1 and b_2 , and the Editor has created the corresponding condition fragments f_1 and f_2 , shown in Figure VII.4b and c respectively. f_1 asks for sensors either having diameter less than the parameter `##DIA` or a protection rating equal to the parameter `##PROT1`, while f_2 asks for sensors having either diameter less than the parameter `##DIA` or width less than the parameter `##WID` so that they fit in a given space. Both condition fragments pass the check of Step 1 of the Construct *CTG* algorithm, since both conjunctions of b_1 and b_2 involve a single variable. In Step 2, structural disjunction operators are introduced to both fragments, shown in Figure VII.4d and e, according to the rules of Figure VII.3. In f_1 , element node `diameter` is under a CHOICE node in the *EST* and element node `protection_rating` is optional. So an OR node is introduced under their lowest common ancestor node `specs`. Similarly, in f_2 , the nodes `diameter` and `width` are both under a CHOICE node in the *EST*, so an OR node is introduced under the node `body_type`.

Step 3 of the Construct *CTG* algorithm just puts f_1 and f_2 together, thus constructing the merged *CTG* shown in Figure VII.6a, where the two fragments are indicated in two different tones of gray.

VII.B.2 Eliminating Redundancies

The Editor eliminates redundancies on the merged *CTG* in order to improve the performance of the generated TQL queries. As shown in [7], efficiency of tree pattern queries depends on the size of the pattern, so it is essential to identify and eliminate redundant nodes. More specifically, according to the rule of Figure VII.5, the Editor renders redundant an element node that has a sibling node labeled with the same variable.

The application of the rule takes time linear to the number of nodes of the *CTG*. The process of eliminating redundant nodes could also be performed on TQL queries, instead of the *CTG*, at run-time. Either way, the final TQL query

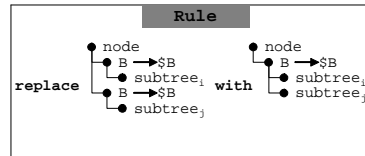
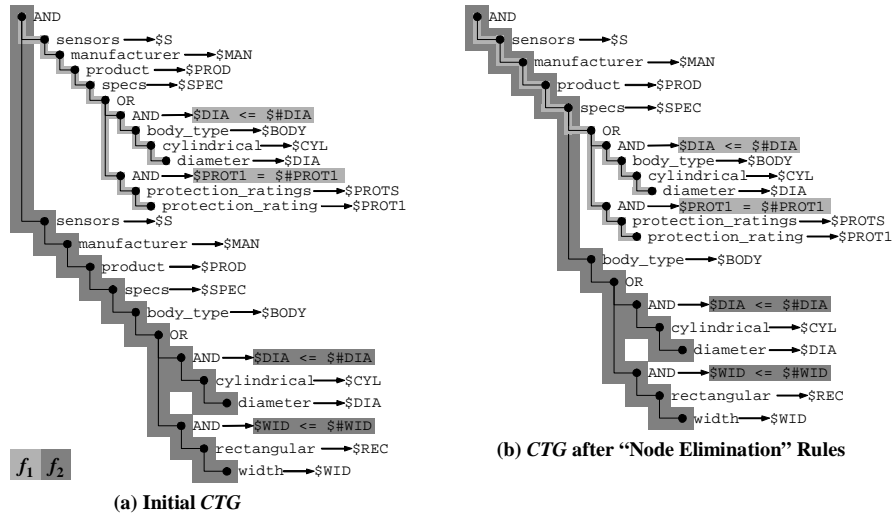


Figure VII.5: “Node Elimination” Rule

Figure VII.6: Eliminating Redundant Nodes on the *CTG*

is the same, so it is preferable to perform the optimization at compile-time.

The rule is eliminating redundancies introduced particularly during the construction of the *CTG*, presented in the previous section. For example, the Construct CTG algorithm constructs the *CTG* of Figure VII.6a by merging two fragments. The path from the `sensors` node to the `specs` node appears in both condition fragments, and every element node along the path is labeled with the same variable in both fragments. One of these paths is eliminated by parsing the *CTG* top-down and iteratively applying the rule of Figure VII.5. The resulting *CTG* is shown in Figure VII.6b. Note that the rule preserves the boundaries of the fragments as element nodes are being eliminated.

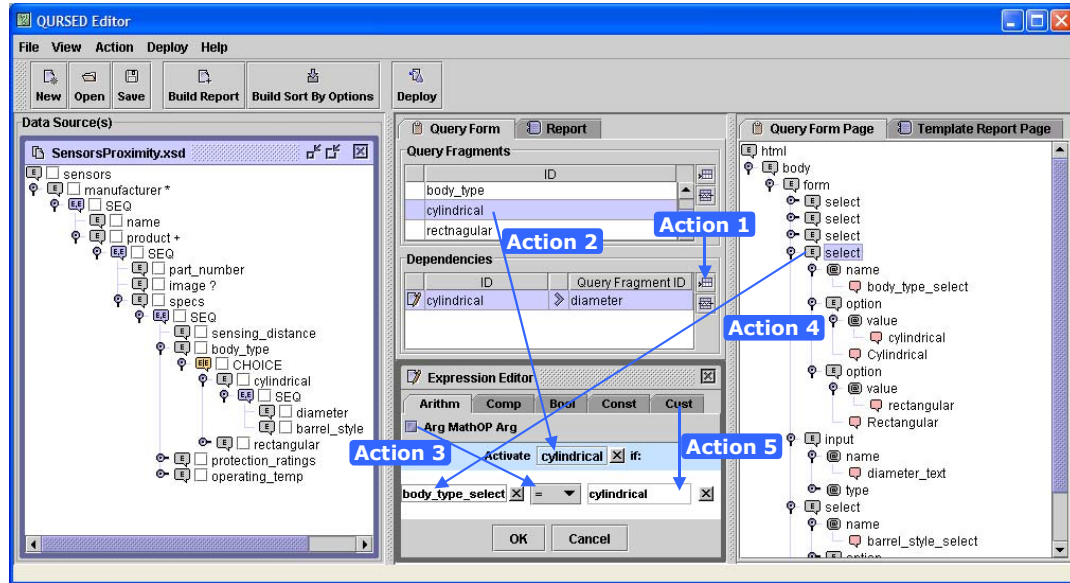


Figure VII.7: Building Dependencies

VII.C Building Dependencies

The Editor provides a set of actions to allow the developer to build a dependency, i.e., to select the dependent condition fragment and to construct the condition of the dependency. As an example, Figure VII.7 demonstrates how the developer builds dependency $d_1 : \langle f_2, \text{\$#BODY} = \text{"cylindrical"}, \{f_1\} \rangle$ of Section VI.D by performing a set of actions indicated by the numbered arrows. Dependency d_1 sets the condition fragment f_2 on the cylindrical dimensions (Figure VI.7a) active if the parameter $\text{\$#BODY}$ is set to “cylindrical”.

First, the developer initiates a dependency (Action 1 of Figure VII.7) and enters a descriptive ID. On the middle panel, a new row appears in the lower table that lists the dependencies, and the expression editor opens at the bottom. She sets the dependent condition fragment to be the “cylindrical” one (Action 2), and builds the condition of the dependency in the expression editor (Action 3). She specifies that the left operand of the equality predicate is a parameter bound to the “Body Type” `select` form control (Action 4), and the right operand to be the string constant “cylindrical” (Action 5). Note that only constant values

The screenshot shows a web browser window titled "Automatic Report Page - Microsoft Internet Explorer". The page displays a report with a table structure. The main table has columns: Name, Part Number, Image, Sensing Distance, Cylindrical, and Protection Ratings. The "Cylindrical" column is further nested into "Diameter mm" and "Barrel Style". The "Protection Ratings" column is nested into "Protection Rating".


Name	Part Number	Image	Sensing Distance	Cylindrical		Protection Ratings
Turck	A123		11.0	Diameter mm	Barrel Style	Protection Rating
				17	Smooth	NEMA1 NEMA3
	B123		25.0	Rectangular		Protection Rating
				Height mm	Width mm	Protection Rating
				10	30	NEMA3 NEMA4

Figure VII.8: Schema-Driven Constructed Report Page

and parameters that bind to form elements can be used in the condition of the dependency, as defined in Section VI.D.

VII.D Building Result Tree Generators

The Editor provides two options for the developer to build the result tree generator *RTG* component of a query set specification, each one associated with a set of corresponding actions. For the first (and simpler) option, called *schema-driven*, the developer only specifies which element nodes of the *EST* she wants to present on the report page. Then, the Editor automatically builds a result tree generator that creates report pages presenting the source data in the form of XHTML tables that are nested according to the nesting of the *EST*. If the developer wants to structure the report page in a different way than the one the *EST* dictates, the Editor provides a second option, called *template-driven*, where the developer provides as input a template report page to guide the result tree generator construction. Both options are described next.

VII.D.1 Schema-Driven Construction of Result Tree Generator

The developer can automatically build a result tree generator based on the nesting of the *EST*. For example, Figure VII.8 shows a report page created from the result tree generator for the data set and the *EST* of Figure V.2. The

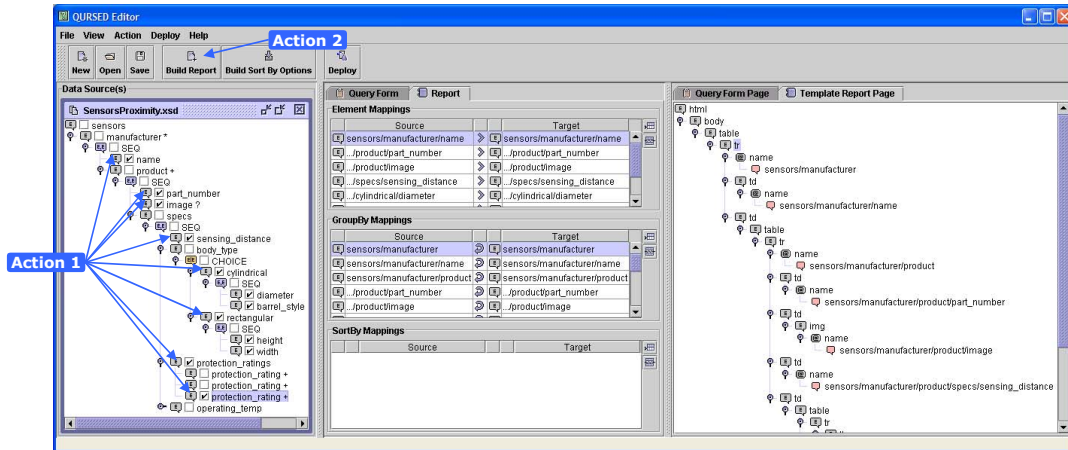
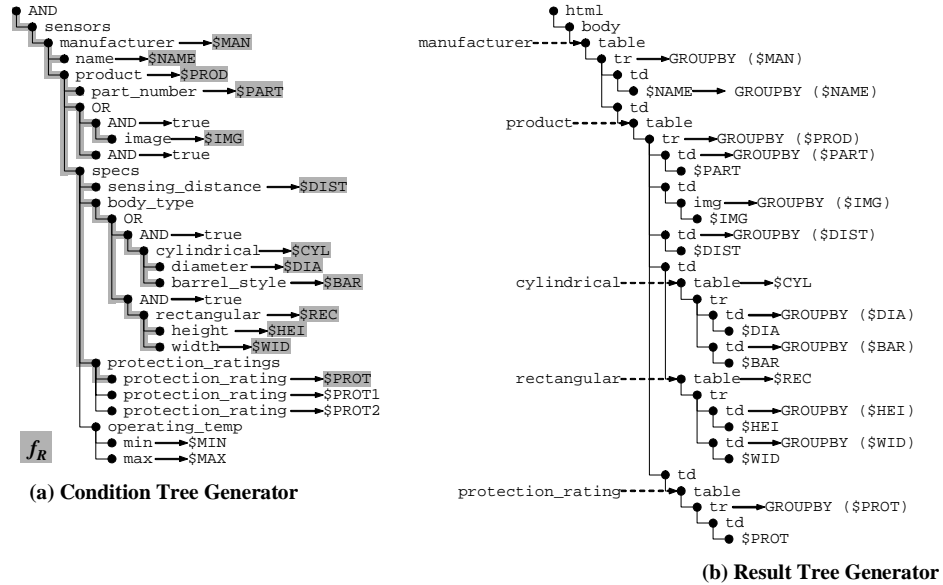


Figure VII.9: Selecting Elements Nodes and Constructing Template Report Page

creation of the result tree generator and the template report page is accomplished by performing the two actions that are indicated by the numbered arrows on the Editor's window of Figure VII.9.

First, the developer uses the checkboxes that appear next to the element nodes of the *EST* to select the ones she wants to present on the report page (Action 1 of Figure VII.9). This action sets the *report* property of the selected element nodes in the *EST* to *true* and constructs the result fragment f_R indicated in the condition tree generator of Figure VII.10a. The variables that will be used in the result tree generator are also indicated. Then, the Editor automatically generates the template report page (Action 2) displayed on the right panel of Figure VII.9 as a tree of XHTML element nodes. Figure VII.10c shows how a WYSIWYG XHTML editor renders the template report page. The Editor translates the above actions into a *QSS* as follows.

In Action 2, the Editor automatically generates the result tree generator of Figure VII.10b that presents the element nodes selected in Action 1 using XHTML `table` element nodes that are nested according to the nesting of the *EST*. For illustration purposes, each `table` element node in Figure VII.10b is annotated with the *EST* element node that it corresponds to. Notice, for example, that the “product” table is nested in the “manufacturer” table, as is the case in the



Name	Part Number	Image	Sensing Distance	Cylindrical		Protection Ratings
../name	../product/part_number	../product/image	../product/specs/sensing_distance	Diameter mm	Barrel Style	Protection Rating
				../body_type/cylindrical/diameter	../body_type/cylindrical/barrel_style	
				Rectangular		../product/specs/protection_ratings/protection_rating
				Height mm	Width mm	
				../body_type/rectangular/height	../body_type/rectangular/width	

(c) Template Report Page

Figure VII.10: Automatically Generated Result Fragment, *RTG* and Template Report Page

EST. The table headers in Figure VII.10c are created from the name labels of the selected element nodes. In the tables, the Editor places the element variables of the element nodes selected in Action 1 as children of `td` (table data cell) element nodes. For example, in the result tree generator of Figure VII.10b the element variable `$NAME` appears as the child of the `td` element node of the “manufacturer” table.

We discuss next how each type of semistructureness of the *EST* is handled by the Editor on the template report page.

Optional Element Nodes: When the developer includes an optional element node in the result, the corresponding result fragment will produce results whether this optional element is or is not present. Figure VII.10a demonstrates the effect of the visual action to select the optional element `image` to appear on the report page.

Repeatable Element Nodes: The Editor handles the repeatable element nodes in the *EST* by automatically generating corresponding `table` elements and group-by lists in the result tree generator. For example, the path from the root of the *EST* to the `name` element node that is selected in Action 1 contains the `manufacturer` repeatable element node, which results in the generation of the “manufacturer” `table` element node, shown in Figure VII.10b, and the group-by list of its `tr` (table row) child element node. This group-by list will generate one table row for each binding of the `$MAN` element variable.

CHOICE Nodes: CHOICE nodes in the *EST* require the Editor to automatically generate OR nodes in the result fragment f_R , as in the case where the CHOICE node above the `cylindrical` and `rectangular` element nodes in the *EST* is translated to an OR node in the result fragment f_R .

The complete algorithm, called *AutoReport*, for constructing the result fragment and the result tree generator, is presented below. The *AutoReport* algorithm inputs the *EST*, where some or all of the element nodes are selected for presentation on the report page, i.e., their *report* property is set to *true*, the result fragment f_R , and proceeds in two steps. The first step manipulates the result fragment f_R by introducing OR nodes based on CHOICE nodes and optional elements in the *EST*. The second step automatically constructs the result tree generator.

The *AutoReport* algorithm assumes the existence of a function $node(\$V_i)$ that given a variable name $\$V_i$ in f_R returns the node n_i of the *EST* that the variable corresponds to. In the case of name variables, $node(\$V_i)$ returns the parent of the node(s) that the name variable corresponds to. It also assumes the existence of a function $copy(n_i)$ that given a node n_i in the *EST* it returns the

copy of it in f_R , if there exists one, or *null*, otherwise.

Algorithm AutoReport

Inputs: The *EST* where some or all of the nodes are selected for presentation on the report page, and the result fragment f_R .

Output: The result fragment f_R and the result tree generator *RTG*.

Method:

Step 1: Manipulation of f_R

// Introduce OR nodes in f_R based on

// CHOICE nodes and optional elements in the *EST*

Traversing f_R top-down, for an element node n_i 1

 If n_i is labeled with a variable $\$V_i$ and $parent(node(\$V_i))$ is a 2

 CHOICE node and $parent(n_i)$ is not an OR node

 If there is a sibling n_j of n_i labeled with a variable $\$V_j$ 3

 such that $node(\$V_j)$ is a sibling of $node(\$V_i)$

 For all sibling element nodes n_j of n_i labeled with a 4

 variable $\$V_j$ such that $node(\$V_j)$ is a sibling of $node(\$V_i)$

 Apply Rule 1 of Figure VII.11 5

 Else 6

 Apply the Rule 2 of Figure VII.11 // Treat n_i as optional element 7

 If n_i is labeled with a variable $\$V_i$ and $node(\$V_i)$ is optional, or n_i is named 8

 with a variable $\$V_i$ and at least one child of $node(\$V_i)$ is optional

 Apply the Rules 2 and 3 of Figure VII.11 correspondingly 9

Step 2: Construction of the result tree generator *RTG*

Create a node n_r named “html”, a node n_b named “body”, 10

a node n_t named “table”, and a node n_{tr} named “tr”

Set n_r as the root of the *RTG*, n_b as a child of n_r , 11

n_t as a child of n_b , and n_{tr} as a child of n_t

Traversing the *EST* top-down and left to right, ignoring SEQ, 12

CHOICE and ALL nodes, for an element node n_i

 BuildTable(n_i, n_{tr}) 13

BuildTable(n_i, n_{tr})

If n_i is either repeatable or $parent(n_i)$ is a CHOICE node 14

 Create a node n_{td} named “td” and a node n_t named “table” 15

 Set n_{td} as a child of n_{tr} and n_t as a child of n_{td} 16

 Create a node named “tr” and set it as the current n_{tr} 17

 If $parent(n_i)$ is a CHOICE node 18

 Attach the Boolean expression $var(n_i)$ to n_t 19

 If n_i is repeatable 20

 Add $var(n_i)$ to the group-by list of n_{tr} 21

If n_i is a selected element node 22

 Create a node n_{th} named “th” and add it as a child of n_{tr} 23

 Create a node named $name(n_i)$ and add it as a child of n_{th} 24

 If n_i is a leaf element node 25

 Create a node named “td”, add it as a child of n_{tr} , 26

 and set it as the current n_{td}

 Create a node named $var(n_i)$ and add it as a child of n_{td} 27

 If $var(n_i)$ is not in any group-by list of an ancestor node 28

 Add $var(n_i)$ to the group-by list of n_{td} 29

For every child element node n_c of n_i 30

 BuildTable(n_c, n_{tr}) 31

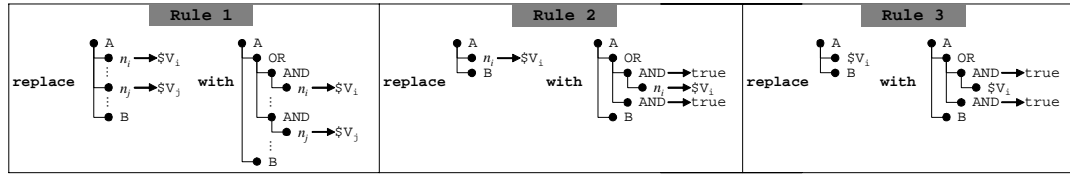


Figure VII.11: “OR Node Introduction” Rules for Result Fragment f_R

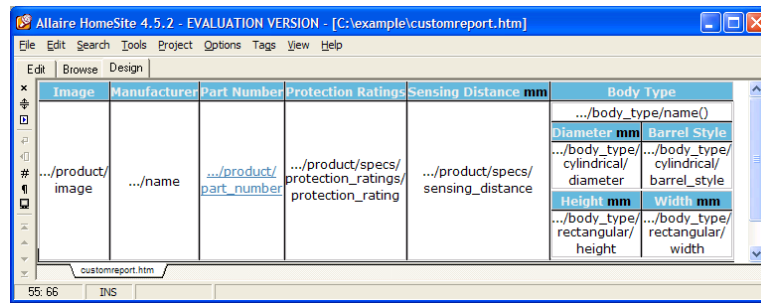


Figure VII.12: Editing the Template Report Page

The result fragment f_R that is manipulated during Step 1 of the AutoReport algorithm is merged with the condition tree generator CTG of a QSS according to Step 3 of the Construct CTG algorithm of Section VII.B.1 and redundant nodes are eliminated using the rule of Figure VII.5.

VII.D.2 Template-Driven Construction of Result Tree Generator

The developer can create more sophisticated report pages and result tree generators by providing to the Editor a template report page she has constructed with an XHTML editor. For example, on the report page of Figure V.3 the developer wants to display the manufacturer’s name for each sensor product, unlike the report page on Figure VII.8 that followed the nesting pattern of the EST , where the **product** is nested in the **manufacturer** element node. To accomplish that, she constructs the template report page shown in Figure VII.12 and provides it to the Editor.

On the right panel of Figure VII.13 the template report page is displayed. Using the EST panel and the template report page panel, the developer constructs

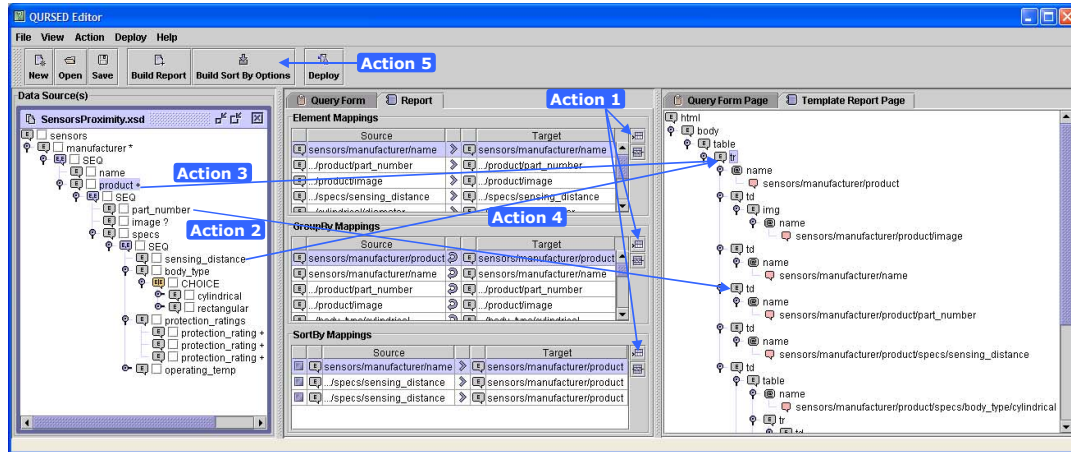


Figure VII.13: Performing Element and Group-By Mappings on the Template Report Page

the result tree generator of the query set specification of Figure VI.5. In particular, the structure of the result tree generator is the structure of the template report page. The rest of the result tree generator (element variables, group-by and sort-by lists) is constructed by performing the actions that are indicated by the numbered arrows on Figure VII.13.

First, the developer creates a new element, group-by or sort-by mapping (Action 1). Depending on what mapping was created, one of Actions 2, 3, or 4 is performed.

In the case of element mapping, the developer drags element nodes from the *EST* and drops them to leaf nodes of the template report page (Action 2). This action places the variable labeling or naming the dragged element node in the result tree generator, and adds the path from the root of the *EST* to the dragged element node to the result fragment f_R . For example, by mapping the `part_number` element node to the `td` element node on the template report page, the `$PART` variable is implicitly placed in the result tree generator of Figure VI.5b.

In the case of group-by mapping, the developer maps element nodes from the *EST* to any nodes of the template report page (Action 3). For example, by mapping the `product` element node to the `tr` element node of the outermost table

in the template report page, the `$PROD` element variable is added to the group-by list of the `tr`. This action will result in one `tr` element node for each binding of the `$PROD` element variable.

The case of sort-by mapping is the same as the group-by mapping, but the developer additionally specifies an optional order. For example, by mapping the `sensing_distance` element node to the `tr` element node of the outermost table, the sort-by list of that element, shown in Figure VI.5b, is generated. The Editor defines automatically a group-by mapping for each sort-by mapping, if one does not exist. Note though that the developer did not specify a fixed order, ascending or descending, thus generating the ordering parameter `$#0_DIST`. This choice allows the end-user to choose the order or exclude `sensing_distance` from the sort-by list altogether.

Finally, the Editor automatically generates and appends the XHTML representation of the “Sort by Options” and “Sort By Selections” drop-down lists to the query form page of Figure V.3 (Action 5). The “Sort by Options” list contains the sort-by mappings defined in Action 4 for which a fixed order has not been specified. The “Sort By Selections” list is initially empty. During run-time, the end-user can select any item from the “Sort by Options”, select “ASC” or “DESC” order, and, using the “+” button, add it to the “Sort By Selections” list. When the end-user submits the query form, the corresponding ordering parameters are instantiated with the order the end-user selected, as explained in the *QSS2TQL* algorithm in Section VI.C.

An engineering benefit from the way the developer builds the result tree generator is that the template report page can easily be opened from any external XHTML editor and further customized visually, even after the mappings have been defined.

Based on the above actions, the result fragment f_R is defined as the set of variables used in the result tree generator that the developer manually constructs. The f_R is constructed by Step 1 of the AutoReport algorithm of Section VII.D.1,

merged with the condition tree generator of a *QSS* according to Step 3 of the Construct *CTG* algorithm of Section VII.B.1, and redundant nodes are eliminated using the rule of Figure VII.5.

VII.E Building Result Boolean Expressions

In Figure V.3, the manufacturer’s column does not display the name as text, but a corresponding image (logo) is presented instead. This effect is accomplished by the three `img` elements, corresponding to the three possible manufacturers, shown in the result tree generator *RTG* of the *QSS* in Figure VI.5 and the Boolean expressions that label them. These expressions are visually defined by the developer on the template report page and are translated by the Editor to Boolean expressions labeling nodes of the *RTG*.

In order to build these Boolean expressions, the Editor provides to the developer a set of actions that is similar to the actions provided for the specification of dependencies as it is presented in Section VII.C. The setting of the Editor is the same with the one in Figure VII.7, except that the “Report” tab is selected in the middle panel and the “Template Report Page” tab is selected in the right panel. The developer builds the Boolean expressions by performing the same set of actions as the ones described in Section VII.C with two differences:

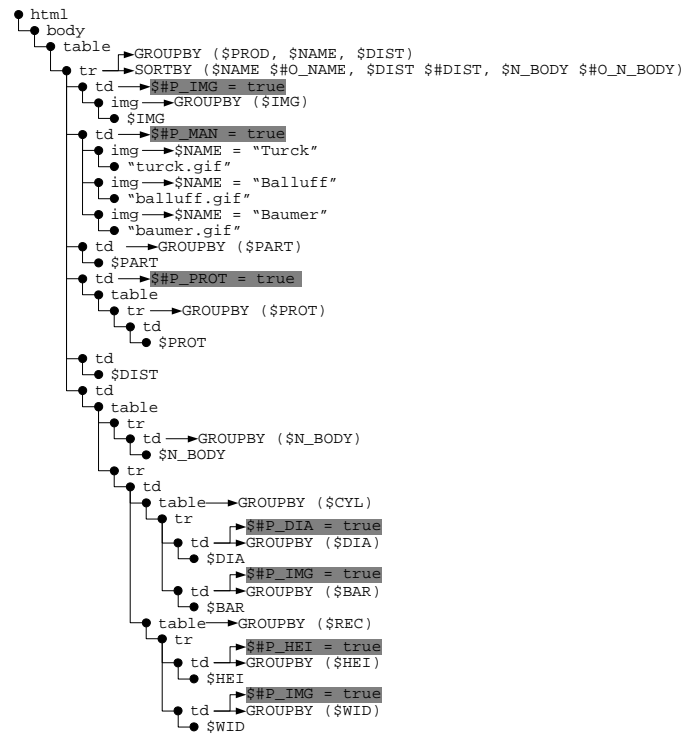
- In Action 2, the developer selects a node from the template report page from the right panel, instead of a condition fragment, to the expression editor’s “Activate” box in Figure VII.7. The subtree rooted at the selected node will be included in the report if the Boolean expression defined in the expression editor evaluates to *true* during run-time.
- In Actions 4 and 5, the developer cannot only specify parameters and constants as operands of the predicates in the Boolean expression, but also any variable, by dragging any element node from the *EST* on the left panel.

The Boolean expressions that the developer defines on the template report page are listed in the “Boolean Expressions” table of the middle panel of Figure VII.13.

Note that the Boolean expressions containing variables are translated to XQuery conditional expressions [14], according to TQL2XQuery algorithm in Appendix F. For example, the three Boolean expressions that label the `img` elements in Figure VI.1b are translated to three conditional expressions, as the XQuery expression in Appendix F shows. If the Boolean expressions contain parameters, then they are evaluated during the formulation of the TQL query, as the *QSS2TQL* algorithm shows in Section VI.C. An example of Boolean expressions containing parameters is given in the next section.

VII.F Dynamic Projection Functionality

On the query form page of Figure V.3, the “Customize Presentation” section allows the end-user to control which columns she wants to project on the report page by selecting the corresponding checkboxes in the “P” column. This *dynamic projection* functionality is provided through the use of Boolean expressions in the result tree generator *RTG* of a *QSS*. Figure VII.14 shows the *RTG* of the *QSS* of Figure VI.5, where Boolean expressions controlling the dynamic projection are labeling `td` (table data cell) element nodes and are indicated with gray shade. These Boolean expressions contain *projection parameters* that start with `$#P_` and correspond to the checkboxes of the “Customize Presentation” section on the query form page of Figure V.3. If a checkbox is checked, then the corresponding Boolean expression evaluates to *true* and the subtree is included in the result tree of the TQL query formulated during run-time. These Boolean expressions are defined by the developer using the actions described in Section VII.E, but instead of nodes from the *EST*, the developer sets as operands of the Boolean expression the checkboxes from the query form page.



Result Tree Generator

Figure VII.14: Boolean Expressions for Dynamic Projection

The above described process assumes that the developer manually constructs the “Customize Presentation” table of Figure V.3. The Editor though has the ability to construct this table automatically as part of the schema-driven construction of the *RTG* described in Section VII.D.1. In this case, the “Customize Presentation” table is constructed according to the nesting of the *EST* just as the template report page is, and is structurally the same as the header row of the template report page. For example, observe that the “Customize Presentation” table on Figure V.3 is structurally the same with the header row of the report page, the only difference being that it is oriented vertically.

More specifically, during Action 2 of Section VII.D.1, the Editor asks the developer if she wants to construct a “Customize Presentation” table. If so, the Editor constructs a table based on the element nodes selected during Action 1 of Section VII.D.1 and lets the developer specify which of them she wants the end-user to be able to include or exclude on the report page. For example, on the “Customize Presentation” table on Figure V.3, the end-user cannot determine the projection of “Part Number” and “Sensing Distance”.

Chapter VIII

Conclusions and Future Work

VIII.A Conclusions

The results of this thesis drastically improve the applicability and usability of integration systems in a wide range of scenarios by defining declarative user-oriented and application-oriented databases interfaces for the domains of the architecture in Figure I.1.

In the Application Domain, I introduced the CLIDE query formulation interface [63] to help a developer, who is building an integration application, avoid a frustrating trial-and-error cycle. CLIDE's architecture consists of a graphical front-end and a back-end. The front-end extends the query builder of Microsoft's SQL Server [81], which is based on the Query-By-Example (QBE) paradigm [95], with a coloring scheme that guides the user toward formulating feasible queries. CLIDE provides guarantees that the suggested query edit actions are complete (i.e. each feasible query can be built by following only suggestions), rapidly convergent (the suggestions are tuned to lead to the closest feasible completions of the query) and suitably summarized (at each interaction step, only a minimal number of actions are suggested). Interaction sessions between the user and the CLIDE front-end are formalized using an *Interaction Graph*, which models the queries as nodes and the actions that the user performs as edges. Consequently, the color of each

action is formally defined as a property of the set of paths that include the action and lead to feasible queries. Then the above guarantees are formally expressed as graph properties. The CLIDE back-end implements a set of algorithms that find a finite set of *closest feasible queries*, related to the current query, and determine the coloring by inspecting it. I provided a set of experiments that illustrate the class of queries and views CLIDE can handle, while maintaining on-line response.

In the Source Domain, I proposed the *Query Set Specification Language (QSSL)* [64] for exporting web services on top of databases which overcomes the limitations of WSDL-based web services by exposing the schema of the underlying source (or view of the source) and a set of supported queries against this schema. QSSL states explicitly the relationship between the input parameters and the output of web services, and the semantic connections the web services have with each other and with the underlying database. QSSL is capable of concisely describing large sets of semantically meaningful parameterized queries, without requiring exhaustive enumeration of them. A QSSL specification is embedded in a WSDL specification, thus extending the current state-of-the-art instead of replacing it, to form a specialized type of web services, called *Data Services*.

In the Web Domain, I developed QURSED, a system for the semi-automatic generation of web-based interfaces for querying and reporting semistructured data. I described the system architecture and the formal underpinnings of the system, including the Tree Query Language for representing semistructured queries, and the succinct and powerful query set specification for encoding the large sets of queries that can be generated by a query form. I described how the tree queries and the query set specification accommodate the needs of query interfaces for semistructured information through the use of condition fragments, OR nodes and dependencies. I also presented the QURSED Editor that allows the GUI-based specification of the interface for querying and reporting semistructured data, and described how the intuitive visual actions result in the production of the query set specification and its association with the visual aspects of the query forms and

reports.

VIII.B Future Work

The next generation of integration scenarios is targeting web communities that need a single point of access to the collection of data owned by a large number of community members without incurring the heavy development and evolution cost of existing integration applications. Each member is responsible of registering her independent data source to a community-shared database, without the need of an administrator/developer who manages the integration process. Moreover, community-users will build applications that query the shared database. I am planning to develop a framework that enables such community-based scenarios and addresses the challenges presented by such dynamic environments, where new sources contribute new content that serves the application needs, while, at the same time, user applications evolve as the community grows.

Community members need tools that assist on several aspects of source registration. They need to easily understand if their source registration contributes to the needs of the applications built on top of the community-shared database, either independently or in conjunction with other registered sources. They also need to know if the content they contribute is consistent with respect to the content of already registered sources and what the degree of redundancy is. To expedite the integration process, schema matching techniques, schema-based and instance-based, can be used to assist owners in identifying common entities and relationships between their source and the community-shared database. On a lower level, an owner might make her contribution by providing interfaces to her sources, such web services, or by providing data that will be warehoused in the community-shared database. In the former case, the owner needs to choose what queries to support, while in the latter, there are issues about data freshness and ownership.

On the other hand, community-users need to be able to build their ap-

plications without requiring a comprehensive overview of all registered sources. CLIDE guides users in formulating executable queries, but more functionality should be offered. Users should be warned if their queries are executable but the result is guaranteed to be empty. They should know the reliability and freshness of the query results they are getting and be able to express their preferences if more than one sources can provide the data they need.

Another research direction that I recently have started to explore is the use of formal workflow languages in implementing database-backed web applications. Workflow languages provide a promising way to express and communicate application specifications. I believe that we can go one step further by turning the workflow specifications into the running code of the web application, by enhancing the workflow specification with actions on the database and presentation of the data. In this way development time will be reduced and the implementation will be faithful to the specification. Another well-known problem is that once applications are built, specifications become outdated and obsolete. In the context of building web applications, using a formal workflow, specific issues will be studied. If the workflow specification changes, which applications queries need to change? If the database schema changes, which applications queries need to change and is the workflow specification violated?

Appendix A

WSDL Specification of a Data Service

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions
  name='FlightsService'
  targetNamespace='http://airline.wSDL/flights/'
  xmlns='http://schemas.xmlsoap.org/wSDL/'
  xmlns:tns='http://airline.wSDL/flights/'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:xsd1='http://airlineQSSX/'
  xmlns:xsd2='http://airlineSchema/'>

  <import
    location='airlineQSSX.xsd'
    namespace='http://airlineQSSX/'/>

  <import
    location='airlineSchema.xsd'
    namespace='http://airlineSchema/'/>

  <message name='queryFlightsRequest'>
    <part name='query' type='xsd1:query'/>
    <part name='result' type='xsd2:result'/>
```

```
</message>
<message name='resultFlightsResponse'>
  <part name='result' type='xsd:result' />
</message>

<portType name='FlightsPortType'>
  <operation name='queryFlights'
    variety='Input-Output'>
    <input message='tns:queryFlightsRequest'
      name='queryFlightsRequest' />
    <output message='tns:resultFlightsResponse'
      name='resultFlightsResponse' />
  </operation>
</portType>
</definitions>
```

Appendix B

QSSX Syntax

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://airlineQSSX/"
  xmlns:qssx = "http://airlineQSSX/">

  <xsd:annotation>
    <xsd:documentation>The root element of a TPX query</xsd:documentation>
  </xsd:annotation>
  <xsd:element name = "query">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref = "qssx:f1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:annotation>
    <xsd:documentation>
      The following element groups correspond to the productions of Figure IV.4d
    </xsd:documentation>
  </xsd:annotation>

  <xsd:group name = "f1"><xsd:sequence>
    <xsd:element name = "step">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name = "identifier" fixed = "flights"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:group>
</xsd:schema>
```

```

<xsd:choice>
  <xsd:sequence>
    <xsd:annotation>
      <xsd:documentation>
        The airline element is chosen as the result node
      </xsd:documentation>
    </xsd:annotation>
    <xsd:element name = 'predicatedExpr'>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name = 'identifier'
            fixed = 'airline' />
          <xsd:group ref = 'qssx:f2' />
          <xsd:element name = 'predicate'
            maxOccurs = 'unbounded'>
            <xsd:complexType><xsd:sequence>
              <xsd:group ref = 'qssx:f3' />
            </xsd:sequence></xsd:complexType>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name='axis'
          use='required'
          type='xsd:string'
          fixed = 'CHILD' />
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:sequence>
    <xsd:annotation>
      <xsd:documentation>
        The flight element is chosen as the result node
      </xsd:documentation>
    </xsd:annotation>
    <xsd:element name = 'step'>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name = 'predicatedExpr'>
            <xsd:complexType><xsd:sequence>
              <xsd:element name = 'identifier'
                fixed = 'airline' />
              <xsd:group ref = 'qssx:f2' />
            </xsd:sequence></xsd:complexType>
          </xsd:element>
          <xsd:group ref = 'qssx:f3'

```

```

maxOccurs = 'unbounded'/'>
</xsd:sequence>
<xsd:attribute name='axis'
use='required'
type='xsd:string'
fixed = 'CHILD'/'>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:choice>
</xsd:sequence>
<xsd:attribute name = 'axis'
use = 'required'
type = 'xsd:string'
fixed = 'CHILD'/'>
</xsd:complexType>
</xsd:element>
</xsd:sequence></xsd:group>

<xsd:group name = 'f2'>
<xsd:sequence>
<xsd:element name = 'predicate'>
<xsd:complexType>
<xsd:sequence>
<xsd:element name = 'function'>
<xsd:complexType>
<xsd:sequence>
<xsd:element name = 'identifier' fixed = 'name'/'>
<xsd:element name = 'constant' type = 'xsd:string'/'>
</xsd:sequence>
<xsd:attribute name='name' fixed='EQUAL'/'>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:group>

<xsd:group name = 'f3'>
<xsd:sequence>
<xsd:element name = 'predicatedExpr'>
<xsd:complexType>
<xsd:sequence>

```

```

        <xsd:element name = 'identifier' fixed = 'flight' />
        <xsd:group ref = 'qssx:f4' minOccurs = '0' />
        <xsd:group ref = 'qssx:f5' />
        <xsd:group ref = 'qssx:f6'
            minOccurs = '0'
            maxOccurs = 'unbounded' />
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:group>

<xsd:group name = 'f4'>
    <xsd:annotation>
        <xsd:documentation>
            Similar to f2 element group
        </xsd:documentation>
    </xsd:annotation>
    ...
</xsd:group>

<xsd:group name = 'f5'>
    <xsd:annotation>
        <xsd:documentation>
            Similar to f2 element group
        </xsd:documentation>
    </xsd:annotation>
    ...
</xsd:group>

<xsd:annotation>
    <xsd:documentation>
        A choice appears in group f6, because leg element might appear as an identifier
        if groups f7, f8 and f9 are not replaced, or as a pedicated expression otherwise
    </xsd:documentation>
</xsd:annotation>
<xsd:group name = 'f6'>
    <xsd:sequence>
        <xsd:element name = 'predicate'>
            <xsd:complexType>
                <xsd:choice>
                    <xsd:element name = 'identifier' fixed = 'leg' />
                    <xsd:sequence>
                        <xsd:element name = 'predicatedExpr'>

```

```

        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name = ''identifier'' fixed = ''leg''/>
            <xsd:group ref = ''qssx:f7'' minOccurs = ''0''/>
            <xsd:group ref = ''qssx:f8'' minOccurs = ''0''/>
            <xsd:group ref = ''qssx:f9'' minOccurs = ''0''/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:choice>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:group>

<xsd:group name = ''f7''>
  <xsd:annotation>
    <xsd:documentation>
      Similar to f2 element group
    </xsd:documentation>
  </xsd:annotation>
  ...
</xsd:group>

<xsd:group name = ''f8''>
  <xsd:annotation>
    <xsd:documentation>
      Similar to f2 element group
    </xsd:documentation>
  </xsd:annotation>
  ...
</xsd:group>

<xsd:group name = ''f9''>
  <xsd:annotation>
    <xsd:documentation>
      Similar to f2 element group
    </xsd:documentation>
  </xsd:annotation>
  ...
</xsd:group>
</xsd:schema>

```


Appendix C

Result XML Schema

```
<?xml version = ‘‘1.0’’ encoding = ‘‘UTF-8’’?>
<xsd:schema xmlns:xsd = ‘‘http://www.w3.org/2001/XMLSchema’’
            targetNamespace=‘‘http://airlineSchema/’’>
  <xsd:element name = ‘‘result’’>
    <xsd:complexType>
      <xsd:choice>
        <xsd:element ref = ‘‘airline’’/>
        <xsd:element ref = ‘‘flight’’/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name = ‘‘airline’’>
    <xsd:annotation>
      <xsd:documentation>
        Element declaration as it appears in the data XML Schema
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>

  <xsd:element name = ‘‘flight’’>...</xsd:element>
```

```
<xsd:element name = 'leg'>...</xsd:element>  
</xsd:schema>
```

Appendix D

TPX Query

```
<query xmlns = 'http://www.db.ucsd.edu/tpx'>
  <step axis = 'CHILD'>
    <identifier>flights</identifier>
    <step axis = 'CHILD'>
      <predicatedExpr>
        <identifier>airline</identifier>
        <predicate>
          <function name = 'EQUALS'>
            <identifier>name</identifier>
            <constant datatype = 'CHARSTRING'>Delta</constant>
          </function>
        </predicate>
      </predicatedExpr>
      <predicatedExpr>
        <identifier>flight</identifier>
        <predicate>
          <function name = 'EQUALS'>
            <identifier>from</identifier>
            <constant datatype = 'CHARSTRING'>
              JFK
            </constant>
          </function>
        </predicate>
      </predicatedExpr>
    </step>
  </step>
</query>
```

```

</predicate>
<predicate>
  <function name = 'EQUALS'>
    <identifier>to</identifier>
    <constant datatype = 'CHARSTRING'>
      LAX
    </constant>
  </function>
</predicate>
<predicate>
  <function name = 'EQUALS'>
    <identifier>day</identifier>
    <constant datatype = 'CHARSTRING'>
      MON
    </constant>
  </function>
</predicate>
<predicate>
  <predicatedExpr>
    <identifier>leg</identifier>
    <predicate>
      <function name = 'EQUALS'>
        <identifier>to</identifier>
        <constant datatype = 'CHARSTRING'>
          LAS
        </constant>
      </function>
    </predicate>
  </predicatedExpr>
</predicate>
</predicatedExpr>
</step>
</step>
</query>

```

Appendix E

XML Schema for TPX Syntax

```
<?xml version = ‘‘1.0’’ encoding = ‘‘UTF-8’’?>
<xsd:schema xmlns:xsd = ‘‘http://www.w3.org/2001/XMLSchema’’
            targetNamespace = ‘‘http://www.db.ucsd.edu/tpx’’>
  <xsd:group name = ‘‘expression’’>
    <xsd:choice>
      <xsd:element ref = ‘‘constant’’/>
      <xsd:element ref = ‘‘function’’/>
      <xsd:element ref = ‘‘predicatedExpr’’/>
      <xsd:element ref = ‘‘step’’/>
      <xsd:element ref = ‘‘identifier’’/>
    </xsd:choice>
  </xsd:group>
  <xsd:element name = ‘‘query’’>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref = ‘‘expression’’/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = ‘‘predicatedExpr’’>
    <xsd:complexType>
      <xsd:sequence>
```

```

        <xsd:group ref = 'expression' />
        <xsd:element ref = 'predicate' maxOccurs = 'unbounded' />
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name = 'predicate'>
    <xsd:complexType>
        <xsd:sequence>
            <xsd:group ref = 'expression' />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name = 'identifier' type = 'xsd:string' />
<xsd:element name = 'constant'>
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base = 'xsd:string'>
                <xsd:attribute name = 'datatype' type = 'xsd:string' />
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
<xsd:element name = 'function'>
    <xsd:complexType>
        <xsd:choice minOccurs = '0' maxOccurs = 'unbounded'>
            <xsd:group ref = 'expression' />
        </xsd:choice>
        <xsd:attribute name = 'name'
            use = 'required'
            type = 'xsd:string'
            fixed = 'EQUAL' />
    </xsd:complexType>
</xsd:element>
<xsd:element name = 'step'>
    <xsd:complexType>

```

```
<xsd:sequence>
  <xsd:group ref = 'expression' />
  <xsd:group ref = 'expression' />
</xsd:sequence>
<xsd:attribute name = 'axis' use = 'required'>
  <xsd:simpleType>
    <xsd:restriction base = 'xsd:NMTOKEN'>
      <xsd:enumeration value = 'CHILD' />
      <xsd:enumeration value = 'DESCENDANT' />
      <xsd:enumeration value = 'SLASHSLASH' />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Appendix F

TQL2XQuery Algorithm

The algorithm TQL2XQuery works on TQL queries, presented in Section VI.A. TQL2XQuery generates an XQuery expression *equivalent* to the input TQL query. The XQuery expressions generated by TQL2XQuery include `GROUPBY` expressions to efficiently perform the groupings. `GROUPBY` expressions are not part of the latest XQuery working draft [14], but the draft includes an issue regarding an explicit `GROUPBY` construct. Such a construct is presented in Appendix G. The choice of XQuery augmented with `GROUPBY` expressions has been made because of the importance of grouping operations for producing nested XML and XHTML output. Explicit `GROUPBY` expressions enable easier optimization of such grouping operations, as is shown in [24]. As Appendix G shows, XQuery+`GROUPBY` expressions can always be translated to XQuery expressions, often of significantly increased complexity: their use results in cleaner query expressions and more opportunities for optimization, but does not affect the generality of the algorithm.

TQL2XQuery inputs a result (sub)tree RT , rooted at n_{RT} , of a TQL query. The algorithm outputs an XQuery expression using nested `FWOR` expressions (`FOR-WHERE-ORDER BY-RETURN`) and element constructors, where `FWOR` expressions are always nested in the `RETURN` clause of their parents. An `FWOR` expression e defines a scope s_e . It follows that scopes are nested. Every variable $\$V$ in the `FOR` clause of an `FWOR` expression e corresponds to a node n in CT , as discussed

in Section VII.B, and we write $n = node(\$V)$ and $\$V = var(n)$. We also write $scope(\$V) = scope(n) = s_e$ to denote the FWOR expression e that $\$V$ is in the scope of. In the algorithm, we represent by S the current scope and by E the current FWOR expression. We define $allvars(S)$ to be the set of all the variables that are in S or in any scope that S is nested in, and we assume that the root N_{CT} of the CT is known to the algorithm.

Initially, the algorithm is called with $TQL2XQuery(N_{RT}, \emptyset, nil)$. N_{RT} is the root of RT for the TQL query under translation. The initial scope is empty, as is the initial FWOR expression.

Algorithm TQL2XQuery

Inputs: An RT node n_{RT} , the current scope S , and the current FWOR expression E .

Output: An XQuery expression equivalent to the input TQL query.

Method:

```

Traverse  $RT$  top-down and left-to-right.
For an element node  $n_{RT}$  of  $RT$ :
Set  $V$  variables in the group-by list  $G$  of  $n_{RT}$  1
 $n_{RT} \leftarrow$  variables in the attached Boolean expression  $b$  of  $n_{RT}$ 
If  $S$  is empty and  $V$  is not empty then // Top-level group-by list 2
    Create a new FWOR expression  $e$ , with FOR clause: 3
    “FOR  $var(child(N_{CT}))$  IN  $document('source.xml')$ ”
    Set  $E \leftarrow e, S \leftarrow s_e$  4
    For each child  $c_i$  of  $child(N_{CT})$  5
        ApplyConditions( $c_i, V, E, S$ ) 6
    Add conjunctively to WHERE clause of  $E$  the 7
    Boolean expression labeling the root AND node  $N_{CT}$ 
If there exists a variable  $\$V_i \in V$  such that  $\$V_i \notin allvars(S)$  8
    //  $node(\$V_i)$  is under an OR node in  $CT$ 
    Create a new FWOR expression  $e$ , set  $E \leftarrow e, S \leftarrow s_e$  9
For every distinct variable  $\$V_i \in V$  and  $\$V_i \notin allvars(S)$  10
    GenerateResultVariable( $\$V_i, E, S$ ) 11
If the group-by list  $G$  of  $n_{RT}$  is not empty 12
    Add to RETURN clause of  $E$  the expression “GROUPBY  $G$  AS” 13
If  $n_{RT}$  has an attached Boolean expression  $b$  14
    Add to RETURN clause of  $E$  the expression “IF  $b$  THEN” 15
If  $name(n_{RT})$  is a constant 16
    Add to RETURN clause of  $E$  the expression  $\langle name(n_{RT}) \rangle$  17
    For each child  $c_i$  of  $n_{RT}$  18
        TQL2XQuery( $c_i, S, E$ ) 19
    Add to RETURN clause of  $E$  the expression  $\langle /name(n_{RT}) \rangle$  20
Else if  $name(n_{RT})$  is a variable 21
    // then the node is guaranteed to be a leaf node,
    // see Definition 16 in Section VI.A
    Add to RETURN clause of  $E$  the expression “ $\{name(n_{RT})\}$ ” 22
If the sort-by list  $S$  of  $n_{RT}$  is not empty 23

```

Add to ORDER BY clause of E the S list	24
ApplyConditions (n_{CT}, V, E, S)	
// n_{CT} is the current node and V is the top-level group-by list	
If n_{CT} is an OR node, not denoting optional element,	25
and no $var(n_D)$ is in V , where n_D is a descendant of n_{CT}	
For each child AND node n_{AND}	26
Create a new SOME...SATISFIES expression e'	27
For each child c_i of n_{AND}	28
ApplyConditions($c_i, V, e', s_{e'}$)	29
Add conjunctively to SATISFIES clause of e'	30
the Boolean expression labeling n_{AND}	
Add conjunctively to WHERE clause of E the disjunction	31
of all SOME...SATISFIES expressions	
Else If n_{CT} is an OR node and there is a $var(n_D)$ is in V ,	32
where n_D is a descendant of n_{CT}	
For each child AND node n_{AND} having a $var(n_D)$ is in V ,	33
where n_D is a descendant of n_{AND}	
For each child c_i of n_{AND}	34
ApplyConditions(c_i, V, E, S)	35
Add conjunctively to WHERE clause of E the	36
Boolean expression labeling n_{AND}	
Else If n_{CT} is an AND node	37
For each child c_i of n_{AND}	38
ApplyConditions(c_i, V, E, S)	39
Add conjunctively to WHERE clause of E the	40
Boolean expression labeling n_{AND}	
Else	41
GenerateConditionVariable(n_{CT}, E, S)	42
For each child c_i of n_{CT}	43
ApplyConditions(c_i, V, E, S)	44
GenerateConditionVariable (n_{CT}, E, S)	
If n_{CT} is an element node	45
Construct a path expression $pe = var(parent(n_{CT}))/name(n_{CT})$	46
If n_{CT} has a name variable	47
// refers to the XPath's name() function [53]	
Construct a path expression $pe = var(parent(n_{CT}))/name()$	48
Add to FOR/SOME clause of E the variable declaration " $var(n_{CT})$ IN pe "	49
GenerateResultVariable ($\$V, E, S$)	
$B \leftarrow \emptyset$	50
Find in CT the lowest element node ancestor n_{LA} of $node(\$V)$	51
such that, in RT , $var(n_{LA})$ in $allvars(S)$	
Construct a relative path expression pe initially consisting of $var(n_{LA})$	52
Walking down the tree path from n_{LA} to $node(\$V)$,	53
for a node n_{CT} of CT on that path:	
If n_{CT} is an element node	54
Construct a path expression $pe = var(n_{LA}))/name(n_{CT})$	55
Add to FOR clause of E the variable declaration " $var(n_{CT})$ IN pe "	56
If n_{CT} is an AND node with a Boolean expression b	57
Add b to B	58

If n_{CT} has a name variable	59
Construct a path expression $pe = var(n_{LA})/name()$	60
Add to FOR clause of E the variable declaration “ $var(n_{CT})$ IN pe ”	61
Set n_{CT} as n_{LA} and repeat from line 53	62
For every Boolean expression $b_i \in B$	63
For every variable $\$V_i$ used in b_i and not in $allvars(S)$	64
GenerateResultVariable($\$V_i, S, E$)	65
Add to WHERE clause of E the conjunction of the expressions in B	66

Initially, the algorithm produces a FWOR expression e for the top-level group-by list of the RT and applies all conditions that appear in CT . This step ensures that the top-level group-by list, and all subsequent ones, is applied on qualified bindings only. All variables in CT that are not under an OR node are declared in the FOR clause of e . The Boolean expression labeling the root AND node of CT appears in the WHERE clause of e . For each OR node in CT , the algorithm produces a SOME...SATISFIES expression e' . The set of e' expressions are connected using Boolean disjunction and placed in the WHERE clause of e . Nested OR nodes in CT result in nested SOME...SATISFIES expressions. All variables in CT that are under an OR node are declared in the SOME clause of some e' . Boolean expressions labeling AND nodes that are children of an OR node appear in the SATISFIES clause of some e' . This step is implemented in lines 2-7 and the subroutines ApplyConditions and GenerateConditionVariable.

As a second step, the algorithm traverses the result tree depth-first and produces a FWOR expression, nested in the RETURN clause of the enclosing FWOR expression, when it encounters a group-by list containing a variable labeling a node under an OR node in CT (lines 8-9). The FOR clause of the FWOR expression declares the variables in the group-by list by traversing the condition tree (lines 10-11 and subroutine GenerateResultVariable). If the nodes of the result tree have an attached Boolean expression, then an IF...THEN condition expression is added to the RETURN clause of the FWOR expression (lines 14-15). Each node of the result tree either constructs an element or generates element content in the RETURN clause (lines 16-22). Finally, if a node in the result tree has a sort-by list, then an ORDER BY clause is added (lines 23-24.) The complexity of the TQL2XQuery algorithm is

polynomial in the size of the input CT and RT .

The following XQuery expression is generated from the TQL2XQuery algorithm for the TQL query in Figure VI.1. Notice that the algorithm can be enhanced easily to add a `name` attribute to all constructed nodes (on line 14), with the value of the attribute being, for example, the complete path of the node. That would allow us, for example, to name the different `<tr>`, `<td>` and `<table>` elements.

```

<html>
  <body>
    <table>{
      FOR $root IN document('source.xml'),
        $S IN $root/sensors,
        $MAN IN $S/manufacturer,
        $NAME IN $MAN/name,
        $PROD IN $MAN/product,
        $PART IN $PROD/part_number,
        $SPEC IN $PROD/specs,
        $DIST IN $SPEC/sensing_distance,
        $BODY IN $SPEC/body_type,
        $N_BODY IN $BODY/name(),
        $PROTS IN $SPEC/protection_ratings,
        $PROT IN $PROTS/protection_rating,
        $PROT1 IN $PROTS/protection_rating
      WHERE
        $PROT1 = "NEMA3"
        AND ((SOME $CYL IN $BODY/cylindrical,
              $DIA IN $CYL/diameter,
              $BAR IN $CYL/barrel_style
              SATISFIES $DIA <= 20 AND $DIA <= 40)
          OR
          (SOME $REC IN $BODY/rectangular,
            $HEI IN $REC/height,
            $WID IN $REC/width
            SATISFIES $HEI <= 20 AND $WID <= 40))
      ORDER BY $NAME DESCENDING, $DIST
      RETURN
        GROUPBY $PROD AS
        <tr>{
          <td>{
            FOR $IMG IN $PROD/image
            RETURN
              GROUPBY $IMG AS
              <img>{$IMG}</img>
          }</td>,
          <td>{

```

```

IF ($NAME = "Turck") THEN <img>"turck.gif"</img>
IF ($NAME = "Balluff") THEN <img>"balluff.gif"</img>
IF ($NAME = "Baumer") THEN <img>"baumer.gif"</img>
}</td>,
{
GROUPBY $PART AS
<td>{$PART}</td>
},
<td>{
<table>{
  GROUPBY $PROT AS
  <tr>{
    <td>{$PROT}</td>
  }</tr>
}</table>
}</td>,
<td>{$DIST}</td>,
<td>{
<table>{
  <tr>{
    GROUPBY $N_BODY AS
    <td>{$N_BODY}</td>
  }</tr>,
  <tr>{
    <td>{
      FOR $CYL IN $BODY/cylindrical,
      $DIA IN $CYL/diameter,
      $BAR IN $CYL/barrel_style
      WHERE
      $DIA <= 20 AND $DIA <= 40
      RETURN
      GROUPBY $CYL AS
      <table>{
        <tr>{
          GROUPBY $DIA AS
          <td>{$DIA}</td>,
          GROUPBY $BAR AS
          <td>{$BAR}</td>
        }</tr>
      }</table>
    }</td>,
    <td>{
      FOR $REC IN $BODY/rectangular,
      $HEI IN $REC/height,
      $WID IN $REC/width
      WHERE
      $HEI <= 20 AND $WID <= 40
      RETURN
      GROUPBY $REC AS
      <table>{
        <tr>{
          GROUPBY $HEI AS

```

```
        <td>{$HEI}</td>,  
        GROUPBY $WID AS  
        <td>{$WID}</td>  
    }</tr>  
}</table>  
}</td>  
}</tr>  
}</table>  
}</td>  
}</tr>  
}</table>  
</body>  
</html>
```

Appendix G

GROUPBY Proposal

The proposal extends the XQuery syntax with the following GroupBy expressions (productions below extend those in <http://www.w3.org/TR/xquery/#nt-bnf>):

```
Expr ::= Expr 'SORTBY' '(' SortSpecList ')'
      | UnaryOp Expr
      | Expr BinaryOp Expr
      | Variable
      | Literal
      | '.'
      | FunctionName '(' ExprList? ')'
      | ElementConstructor
      | '(' Expr ')'
      | '[' ExprList? ']'
      | PathExpr
      | Expr Predicate
      | FlwrExpr
      | 'IF' Expr 'THEN' Expr 'ELSE' Expr
      | ('SOME' | 'EVERY') Variable 'IN' Expr 'SATISFIES' Expr
      | ('CAST' | 'TREAT') 'AS' Datatype '(' Expr ')'
      | Expr 'INSTANCEOF' Datatype
      | GroupBy
      /***** new *****/
GroupBy ::= 'GROUPBY' VarList? HavingClause? 'AS' Expr
/***** new *****/
VarList ::= Variable (',' VarList)?
/***** new *****/
HavingClause ::= 'HAVING' Expr
/***** new *****/
```

The rest of the grammar remains unchanged. A GroupBy expression returns an unordered collection. The example below refers to the “Use Case XMP” DTD and data (in <http://www.w3.org/TR/xmlquery-use-cases>).

EXAMPLE Grouping elements in the returned document. “For each author, return the number of book titles she published, as well as the list of those titles and their year of publication”.

```
FOR $b IN document('http://www.bn.com')/bib/book,
  $a IN $b/author,
  $t IN $b/title,
  $y IN $b/@year
RETURN
GROUPBY $a AS
<result> $a,
  <number> count(distinct($t)) </number>,
  GROUPBY $t, $y AS
  <titleYear>
    $t,
    <year> $y </year>
  </titleYear>
</result>
```

Notice how the same variable `$t` can be used both outside a `GROUPBY` and inside a `GROUPBY`. Outside the `GROUPBY` its value is a collection, inside the `GROUPBY` its value is a node. The same query can be expressed without `GROUPBY` as follows. Here we have to construct an intermediate collection only to apply ‘distinct’ to it and then to iterate over it:

```
FOR $a IN distinct(document('http://www.bn.com')/bib/book/author)
LET $t = document('http://www.bn.com')/bib/book[author=$a]/title
RETURN
<result> $a
  <number> count(distinct($t)) </number>
  FOR $Tup IN distinct(
    FOR $b IN document('http://www.bn.com')/bib/book[author=$a],
      $t IN $b/title,
      $y IN $b/@year
    RETURN <Tup> <t> $t </t> <y> $y </y> </Tup>),
    $t IN $Tup/t/node(),
    $y IN $Tup/y/node()
  RETURN
  <titleYear>
    $t,
    <year> $y </year>
  </titleYear>
</result>
```


Bibliography

- [1] MDA Guide Version 1.0.1. Object Management Group, Inc., 2003. [Http://www.omg.org/docs/omg/03-06-01.pdf](http://www.omg.org/docs/omg/03-06-01.pdf).
- [2] Amazon E-Commerce Service 4.0. [Http://www.amazon.com/gp/aws/sdk/](http://www.amazon.com/gp/aws/sdk/).
- [3] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web*. Morgan Kaufman, San Francisco, CA, 2000.
- [4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [5] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Representing and Querying XML with Incomplete Information. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2001.
- [6] Jürgen Albert, Dora Giammarresi, and Derick Wood. Normal form algorithms for extended context-free grammars. *Theor. Comput. Sci.*, 267(1–2):35–47, 2001.
- [7] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of Tree Pattern Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.
- [8] Google Web APIs. [Http://www.google.com/apis/](http://www.google.com/apis/).
- [9] Microsoft ASP.NET. Microsoft Corporation, 2005. [Http://www.asp.net/](http://www.asp.net/).
- [10] Paolo Atzeni, Giansalvatore Mecca, and Paolo Merialdo. To Weave the Web. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 206–215, 1997.
- [11] Philip A. Bernstein, Michael L. Brodie, Stefano Ceri, David J. DeWitt, Michael J. Franklin, Hector Garcia-Molina, Jim Gray, Gerald Held, Joseph M. Hellerstein, H. V. Jagadish, Michael Lesk, David Maier, Jeffrey F. Naughton, Hamid Pirahesh, Michael Stonebraker, and Jeffrey D. Ullman. The Asilomar Report on Database Research. *SIGMOD Record*, 27(4):74–80, 1998.
- [12] Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, and Christoph Quix. Industrial-Strength Schema Matching. *SIGMOD Record*, 33(4):38–43, 2004.

- [13] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. W3C Recommendation 28 October 2004, 2004. [Http://www.w3.org/TR/xmlschema-2/](http://www.w3.org/TR/xmlschema-2/).
- [14] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Working Draft 11 February 2005, 2005. [Http://www.w3.org/TR/xquery/](http://www.w3.org/TR/xquery/).
- [15] David Booth and Canyang Kevin Liu. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. W3C Working Draft 21 December 2004, 2004. [Http://www.w3.org/TR/wsdl20-primer/](http://www.w3.org/TR/wsdl20-primer/).
- [16] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular Tree and Regular Hedge Languages over Unranked Alphabets: Version 1. *HKUST Theoretical Computer Science Center Research Report*, HKUST-TCSC-2001-05, 2001.
- [17] Michael J. Carey, Laura M. Haas, Vivekananda Maganty, and John H. Williams. PESTO : An Integrated Query/Browser for Object Databases. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 203–214, 1996.
- [18] Sudarshan S. Chawathe, Thomas Baby, and Jihwang Yeo. VQBD: Exploring Semistructured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.
- [19] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your Mediators Need Data Conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 177–188, 1998.
- [20] Sara Cohen, Yaron Kanza, Yakov A. Kogan, Werner Nutt, Yehoshua Sagiv, and Alexander Serebrenik. EquiX - Easy Querying in XML Databases. In *Proceedings of the 2nd International Workshop on the Web and Databases*, pages 43–48, 1999.
- [21] Macromedia ColdFusion. Macromedia, Inc., 2003. [Http://www.macromedia.com/software/coldfusion/](http://www.macromedia.com/software/coldfusion/).
- [22] Sara Comai, Ernesto Damiani, and Piero Fraternali. Computing graphical queries over XML data. *ACM Trans. Inf. Syst.*, 19(4):371–430, 2001.
- [23] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. XML-QL. In *W3C Query Languages Workshop*, 1998.
- [24] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT Logical Framework for XQuery. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 168–179, 2004.
- [25] Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft 11 February 2005, 2005. [Http://www.w3.org/TR/xquery-semantics/](http://www.w3.org/TR/xquery-semantics/).

- [26] Denise Draper, Alon Y. Halevy, and Daniel S. Weld. The Nimble Integration Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.
- [27] Micah Dubinko, Leigh L. Klotz, Roland Merrick, and T. V. Raman. XForms 1.0. W3C Recommendation 14 October 2003, 2003. [Http://www.w3.org/TR/2003/REC-xforms-20031014/](http://www.w3.org/TR/2003/REC-xforms-20031014/).
- [28] Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive Query Plans for Data Integration. *Journal of Logic Programming*, 43(1), 2000.
- [29] Andrew Eisenberg and Jim Melton. SQL/XML is Making Good Progress. *SIGMOD Record*, 31(2):101–108, 2002.
- [30] Anat Eyal and Tova Milo. Integrating and customizing heterogeneous e-commerce applications. *VLDB Journal*, 10(1):16–38, 2001.
- [31] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. W3C Recommendation 28 October 2004, 2004. [Http://www.w3.org/TR/xmlschema-0/](http://www.w3.org/TR/xmlschema-0/).
- [32] Mary Fernandez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft 11 February 2005, 2005. [Http://www.w3.org/TR/xpath-datamodel/](http://www.w3.org/TR/xpath-datamodel/).
- [33] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. Declarative Specification of Web Sites with Strudel. *VLDB Journal*, 9(1):38–55, 2000.
- [34] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient Evaluation of XML Middle-ware Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.
- [35] BEA Liquid Data for WebLogic. BEA Systems, Inc., 2005. [Http://bea.com](http://bea.com).
- [36] Piero Fraternali. Tools and Approaches for Developing Data-Intensive Web Applications: A Survey. *ACM Comput. Surv.*, 31(3):227–263, 1999.
- [37] Piero Fraternali and Paolo Paolini. Model-driven development of Web applications: the AutoWeb system. *ACM Trans. Inf. Syst.*, 18(4):323–382, 2000.
- [38] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.
- [39] Michael Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufman, San Mateo, CA, 1987.
- [40] Roy Goldman and Jennifer Widom. Interactive Query and Search in Semi-structured Databases. In *Proceedings of the 1st International Workshop on the Web and Databases*, pages 52–62, 1998.
- [41] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 276–285, 1997.

- [42] Alon Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 10(4):270–294, 2001.
- [43] Macromedia HomeSite. Macromedia, Inc., 2003. [Http://www.macromedia.com/software/homesite/](http://www.macromedia.com/software/homesite/).
- [44] Grant Hutchison. DB2 Web Services, 2001. [Http://www-106.ibm.com/developerworks/db2/zones/webservices/bigpicture.html](http://www-106.ibm.com/developerworks/db2/zones/webservices/bigpicture.html).
- [45] Microsoft Visual InterDev. Microsoft Corporation, 2003. [Http://msdn.microsoft.com/vinterdev/](http://msdn.microsoft.com/vinterdev/).
- [46] Michael Kay. XSL Transformations (XSLT) Version 2.0. W3C Working Draft 11 February 2005, 2005. [Http://www.w3.org/TR/xslt20/](http://www.w3.org/TR/xslt20/).
- [47] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
- [48] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovic, Raghu Ramakrishnan, Yirong Xu, and David DeWitt. Limiting Disclosure in Hippocratic Databases. In *VLDB*, pages 108–119, 2004.
- [49] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 251–262, 1996.
- [50] Alon Y. Levy, Anand Rajaraman, and Jeffrey D. Ullman. Answering Queries Using Limited External Processors. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–237, 1996.
- [51] Chen Li and Edward Y. Chang. Answering Queries with Useful Bindings. *ACM Transactions on Database Systems (TODS)*, 26(3), 2001.
- [52] Bertram Ludäscher, Yannis Papakonstantinou, and Pavel Velikhov. Navigation-Driven Evaluation of Virtual Mediated Views. In *Proceedings of the 7th International Conference on Extending Database Technology*, pages 150–165, 2000.
- [53] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft 11 February 2005, 2005. [Http://www.w3.org/TR/xpath-functions/](http://www.w3.org/TR/xpath-functions/).
- [54] Jim Melton and Subramanian Muralidhar. XML Syntax for XQuery 1.0 (XQueryX). W3C Working Draft 4 April 2005, 2005. [Http://www.w3.org/TR/xqueryx/](http://www.w3.org/TR/xqueryx/).
- [55] Kuassi Mensah and Ekkehard Rohwedder. Oracle9i Database Web Services. White Paper, 2002. [Http://www.oracle.com](http://www.oracle.com).

- [56] Gerome Miklau and Dan Suciu. Containment and Equivalence for an XPath Fragment. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 65–76, 2002.
- [57] Kevin D. Munroe and Yannis Papakonstantinou. BBQ: A Visual Interface for Integrated Browsing and Querying of XML. In *Proceedings of the 5th Working Conference on Visual Database Systems*, pages 277–296, 2000.
- [58] Alan Nash and Bertram Ludäscher. Processing Unions of Conjunctive Queries with Negation under Limited Access Patterns. In *EDBT*, 2004.
- [59] Jakob Nielsen. *Designing Web Usability*. New Riders Publishing, 2000.
- [60] JavaServer Pages. Sun Microsystems, Inc., 2005. [Http://java.sun.com/products/jsp/](http://java.sun.com/products/jsp/).
- [61] Yannis Papakonstantinou, Vinayak R. Borkar, Maxim Orgiyan, Konstantinos Stathatos, Lucian Suta, Vasilis Vassalos, and Pavel Velikhov. XML queries and algebra in the Enosys integration platform. *Data Knowl. Eng.*, 44(3):299–322, 2003.
- [62] Yannis Papakonstantinou, Michalis Petropoulos, and Vasilis Vassalos. QURSED: Querying and Reporting Semistructured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 192–203, 2002.
- [63] Michalis Petropoulos, Alin Deutsch, and Yannis Papakonstantinou. Interactive Query Formulation for Systems that Rewrite Queries Using Views. Available online at <http://www.db.ucsd.edu/People/michalis/pubs/clide.pdf>.
- [64] Michalis Petropoulos, Alin Deutsch, and Yannis Papakonstantinou. Query Set Specification Language (QSSL). In *WebDB*, pages 99–104, 2003.
- [65] Michalis Petropoulos, Yannis Papakonstantinou, and Vasilis Vassalos. Building XML query forms and reports with XQForms. *Computer Networks*, 39(5):541–558, 2002.
- [66] Michalis Petropoulos, Yannis Papakonstantinou, and Vasilis Vassalos. Graphical Query Interfaces for Semistructured Data: The QURSED System. *ACM Transactions on Internet Technology*, 5(2), 2005.
- [67] Michalis Petropoulos, Vasilis Vassalos, and Yannis Papakonstantinou. XML Query Forms (XQForms): Declarative Specification of XML Query Interfaces. In *Proceedings of the 10th International World Wide Web Conference*, pages 642–651, 2001.
- [68] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. Translating Web Data. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 598–609, 2002.
- [69] Rachel Pottinger and Alon Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2-3), 2000.

- [70] Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, and Jennifer Widom. (Querying Semistructured Heterogeneous Information). In *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases*, pages 319–344, 1995.
- [71] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 Specification. W3C Recommendation 24 December 1999, 1999. [Http://www.w3.org/TR/html4/](http://www.w3.org/TR/html4/).
- [72] Erhard Rahm and Philip A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, 2001.
- [73] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering Queries Using Templates with Binding Patterns. In *PODS*, pages 105–112, 1995.
- [74] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *SIGMOD*, pages 551–562, 2004.
- [75] Mary Tork Roth and Peter M. Schwarz. Don’t Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB*, pages 266–275, 1997.
- [76] Lawrence A. Rowe. ”Fill-in-the-Form” Programming. In *VLDB*, 1985.
- [77] Waqar Sadiq and Sandeep Kumar. Web Service Description Usage Scenarios. W3C Working Draft 4 June 2002, 2002. [Http://www.w3.org/TR/ws-desc-usecases/](http://www.w3.org/TR/ws-desc-usecases/).
- [78] Harald Schöning and Jürgen Wäsch. Tamino - An Internet Database System. In *Proceedings of the 7th International Conference on Extending Database Technology*, pages 383–387, 2000.
- [79] Luc Segoufin and Victor Vianu. Validating Streaming XML Documents. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 53–64, 2002.
- [80] Microsoft BizTalk Server. Microsoft Corporation, 2004. [Http://www.microsoft.com/biztalk/](http://www.microsoft.com/biztalk/).
- [81] Microsoft Corporation. SQL Server. [Http://www.microsoft.com/sql/](http://www.microsoft.com/sql/).
- [82] Web Services and Service-Oriented Architectures. [Http://www.service-architecture.com/](http://www.service-architecture.com/).
- [83] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimón Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 65–76, 2000.

- [84] Abraham Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman. Database Systems: Achievements and Opportunities - The “Lagunita” Report of the NSF Invitational Workshop on the Future of Database System Research held in Palo Alto, California, February 22-23, 1990. *SIGMOD Record*, 19(4):6–22, 1990.
- [85] Microsoft SQL Server 2000 Web Services Toolkit. Microsoft Corporation, 2002. [Http://www.microsoft.com/sql/techinfo/xml/default.asp](http://www.microsoft.com/sql/techinfo/xml/default.asp).
- [86] TIBCO XML Transform. TIBCO Software Inc., 2005. [Http://tibco.com](http://tibco.com).
- [87] Edward R. Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, 1997.
- [88] Macromedia Dreamweaver UltraDev. Macromedia, Inc., 2003. [Http://www.macromedia.com/software/ultradev/](http://www.macromedia.com/software/ultradev/).
- [89] Vasilis Vassalos and Yannis Papakonstantinou. Describing and Using Query Capabilities of Heterogeneous Sources. In *VLDB*, 1997.
- [90] Vasilis Vassalos and Yannis Papakonstantinou. Expressive Capabilities Description Languages and Query Rewriting Algorithms. *Journal of Logic Programming*, 43(1):75–122, 2000.
- [91] BEA WebLogic Workshop. BEA Systems, Inc., 2005. [Http://bea.com](http://bea.com).
- [92] Oracle XSQL. Oracle, 2004. [Http://www.oracle.com](http://www.oracle.com).
- [93] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *VLDB*, pages 82–94, 1981.
- [94] Ramana Yerneni, Chen Li, Hector Garcia-Molina, and Jeffrey Ullman. Computing Capabilities of Mediators. In *SIGMOD*, pages 443–454, 1999.
- [95] Moshé M. Zloof. Query By Example. In *Proceedings of the AFIPS Conference*, pages 431–438, 1975.