Models and Methods for Advanced Web Applications and Social Networks Automation

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Submitted to the Senate of the Technion - Israel Institute of Technology

Shebat 5772 Haifa January 2012
This research was carried out under the supervision of

Prof. Oded Shmueli

Faculty of Computer Science

The generous financial help of the Technion, the Department of Computer Science and the Jacobs fund is gratefully acknowledged.
Acknowledgements

This work of research was carried out under the supervision of Prof. Oded Shmueli, in the Department of Computer Science at the Technion.

I am deeply grateful to Oded for guiding me through this research with endless energies, for the encouragement and invaluable advice that he always had for me, for having time and patience for me despite so many other duties and for teaching me so much. Oded, I could never thank you enough for believing in me.

I wish to thank my friends at the Technion, Eitan Azriel, Hanna Mazzawi, Keren Censor-Hillel and Mirit Shalem, for their support throughout the years.

I specially thank my parents, Kishuta and David, for their unconditional love and support. This thesis is dedicated to them.
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1.1   Overview of the Thesis</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1.2   Datalog and Semi-Structured Data</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1.3   Query Networks</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>1.4   Protocols for Social Networks</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1.5   The SoQL Language</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Datalog and Semi-Structured Data</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2.1   Introduction</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2.1.1 Chapter Outline</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2.2   Data Model</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2.2.1 Basic Concepts</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2.2.1.1 Datalog</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2.2.1.2 XML and XPath</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2.2.1.3 XPath\textsuperscript{L} - Introducing Predicates for XML processing</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>2.2.1.4 Values</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2.2.2 Object Oriented Data Model</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2.2.2.1 Objects</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2.2.2.2 XML Trees</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>2.2.2.3 Relations</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>2.2.2.4 Isomorphism of Nodes</td>
<td>19</td>
</tr>
</tbody>
</table>
2.2.2.5 Value Objects ................................................. 20
2.2.3 XPath Examples ................................................. 20
  2.2.3.1 Same Item Categories Example ......................... 20
  2.2.3.2 Top Selling Items Example .............................. 22
  2.2.3.3 Basketball Example ...................................... 24
  2.2.3.4 More On Values .......................................... 26
2.2.4 The Database .................................................. 28
  2.2.4.1 EDB Relations ............................................ 28
  2.2.4.2 The set E_d of Allowed Expressions .................... 29
2.2.5 XPath Expressions Returning String, Boolean or Number ................. 30
2.3 System, Evaluation Approaches and Algorithms .......................... 31
  2.3.1 Two Evaluation Approaches ................................. 31
  2.3.2 The Static Approach ....................................... 32
  2.3.3 On-Demand for Conjunctive Queries ....................... 33
    2.3.3.1 Column-Based On-Demand ............................... 34
    2.3.3.2 Join-Based On-Demand ................................. 37
    2.3.3.3 Quick On-Demand Evaluation ......................... 40
    2.3.3.4 Conjunctive Queries with Negated predicates .......... 44
  2.3.4 On-Demand for Recursive Queries .......................... 45
2.4 System Prototype, Algorithm Implementation and Experimentation ............ 50
  2.4.1 System Prototype ........................................... 50
  2.4.2 Implementation and Experimentation ....................... 52
    2.4.2.1 Data and Queries ..................................... 52
    2.4.2.2 Results and Discussion ............................... 58
  2.4.3 Recursive Queries .......................................... 59
  2.4.4 Experiments with Populating Val .......................... 60
2.5 Theoretical Results .............................................. 61
  2.5.1 Undecidability of XPath^L Satisfiability ................. 61
  2.5.2 Identifying all Distinct Values in a Tree .................. 63
  2.5.3 XPath^L dialects Expressiveness Results .................. 65
2.5.4 $L \equiv L_v$ .................................................. 66
2.5.5 $L_c \prec L$ .................................................. 67
2.5.6 $L^+ \equiv L_c^+$ ............................................. 68

2.6 Conjunctive queries with XPath-inspired predicates over DAGs .......................... 69
   2.6.1 Motivation .................................................. 69
   2.6.2 Complexity of Conjunctive Queries over Trees ........................................ 71
   2.6.3 Results ..................................................... 72
   2.6.4 Definitions ................................................ 72
      2.6.4.1 Generalization to DAGs. ................................ 73
   2.6.5 Complexity of conjunctive queries over trees ........................................ 74
      2.6.5.1 Polynomial Results. .................................. 74
      2.6.5.2 NP-Completeness Results ............................. 75
   2.6.6 Complexity Results for Conjunctive Queries over DAGs ............................. 76
      2.6.6.1 Polynomial Results .................................. 76
      2.6.6.2 NP-Completeness Results ............................. 77
      2.6.6.3 Elimination of multiple labels. ........................ 82

2.7 Related Work .................................................. 82

3 Query Networks .................................................. 85
   3.1 Introduction ................................................ 85
      3.1.1 Motivational Example .................................. 86
      3.1.2 Model .................................................... 87
      3.1.3 Chapter Outline ........................................ 89
   3.2 Preliminaries ................................................ 90
   3.3 Network Evaluation Algorithms ........................................ 94
      3.3.1 The Basic Algorithm and Related Results ......................... 94
         3.3.1.1 The Preservation of Cycles Property .................... 95
         3.3.1.2 One-Round Evaluation of DAG Networks ................. 96
      3.3.2 The Backward-Radius Triggering Algorithm ..................... 97
      3.3.3 The Divide and Conquer Algorithm (DAC) ....................... 99
   3.4 Extensions ................................................... 101
3.5 Implementation and Experimentation .............................................. 104
  3.5.1 Synthetic Datasets .......................................................... 105
  3.5.2 DBLP Datasets ............................................................... 107
3.6 Query Networks with Proposal and Acceptance Queries .................... 110
  3.6.1 Example ................................................................. 111
  3.6.2 Formalization ............................................................. 112
3.7 Evaluation Algorithms for Networks with Acceptance Queries .......... 115
  3.7.1 Basic-a Evaluation Algorithm ........................................... 116
  3.7.2 Propose-Accept Evaluation Algorithm ................................. 117
  3.7.3 Propose-Accept with Backward-Radius Triggering .................. 117
  3.7.4 Divide and Conquer for Networks with Acceptance Queries ....... 119
3.8 Experimentations with Acceptance Queries .................................. 121
3.9 Theoretical Results ............................................................. 127
  3.9.1 The Network Fixpoint Problem. ....................................... 127
  3.9.2 The Query Substitution Problem ...................................... 128
  3.9.3 The k-Popularity Problem. ........................................... 130
  3.9.4 The Edge-Creating Query Assigning Problem. ..................... 133
  3.9.5 Model Variants ......................................................... 134
3.10 Related Work ................................................................. 135

4 Protocols for Social Networks .................................................... 137
  4.1 Introduction ................................................................. 137
    4.1.1 Directed Networks and Communication ................................. 138
    4.1.2 Chapter Outline ...................................................... 139
  4.2 A Protocol for Status Determination ..................................... 139
    4.2.1 Examples ............................................................. 140
    4.2.2 Protocol Messages .................................................. 140
    4.2.3 Modifiers ............................................................ 141
    4.2.4 Politeness ........................................................... 143
    4.2.5 Structure of Messages ............................................... 144
## List of Figures

2.1 Supermarket purchases XML document. Text nodes are in *italics*. ........................................ 21
2.2 Supermarket weekly top-3 items XML document. Text nodes are in *italics*. ......................... 23
2.3 Basketball games XML document. Text nodes are in *italics*. ............................................... 24
2.4 Prototype’s architecture. ........................................................................................................... 51
2.5 Benchmark Relations and their contents. ................................................................................... 53
2.6 Results for Q1. ............................................................................................................................ 54
2.7 Results for Q2. ............................................................................................................................ 54
2.8 Results for Q3. ............................................................................................................................ 54
2.9 Results for Q4. ............................................................................................................................ 55
2.10 Results for Q5. .......................................................................................................................... 55
2.11 Results for Q6. XQuery did not return in reasonable time. ....................................................... 55
2.12 Results for Q7. .......................................................................................................................... 56
2.13 Over Populating Factor values. ................................................................................................. 56
2.14 Queries used in the experiments (a). ......................................................................................... 56
2.15 Queries used in the experiments (b). ......................................................................................... 57
2.16 Results for query Q8. .................................................................................................................. 60
2.17 Results for query Q9. .................................................................................................................. 60
2.18 Results of experiments with populating *Val*. ........................................................................... 61
2.19 (a) Electronic bookstore ontology. (b) Event pattern expressed in the conjunctive
query. A straight arrow represents direct causality, and a curly arrow represents
causality (possibly) through other events. ....................................................................................... 70
2.20 Tree used for proving NP-Hardness of \( \{ Child(\cdot,\cdot), Child^+(\cdot,\cdot), Label_\alpha(\cdot)_{\alpha \in \Sigma}\} \)
over trees. ........................................................................................................................................ 75
2.21 DAG for proving NP-Hardness of \{Child^+(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}\}. 78

2.22 DAG for proving NP-Hardness of \{NextSibling(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}\}. 81

3.1 The network for the example in Section 3.6.1. 87

3.2 The network after adding connections. 88

3.3 The query graph corresponding to the query \(F(n, X) \leftarrow F(n, Y), F(Y, X), F(n, Z), F(Z, X)\). 91

3.4 Query Network Example. 92

3.5 The network, after the additions. 93

3.6 Divide and Conquer Example. 102

3.7 Illustration of the Synthetic Datasets Structure. 105

3.8 Time results of Experiment 1. Datasets ordered by IDB/EDB ratio. 106

3.9 Time results of Experiment 1. Datasets ordered by IDB size. 107

3.10 Results of Experiment 1. Datasets ordered by EDB size. 107

3.11 Results of Experiment 2. 108

3.12 Results of Experiment 3. 108

3.13 Results of Experiment 4. 109

3.14 Results of Experiment 5. 109

3.15 The Tennis Players Network and players’ data. Solid edges are original edges, and dashed edges are edges added by evaluation. 112

3.16 (1) \(n\)’s proposal query graph. (2) \(n\)’s acceptance query graph. 115

3.17 Results of Experiments 1. 123

3.18 Results of Experiments 2. 124

3.19 Results of Experiments 3. 124

3.20 Results of Experiments 4. 125

3.21 Results of Experiments 5. 125

3.22 Results of Experiments 6. 126

3.23 Results of Experiments 7. 126

3.24 Results of Experiments 8. 127

3.25 The query used in the reduction to the Network Fixpoint Problem. 129

3.26 The query substitution instance for the corresponding reduction. 131

3.27 The k-Popularity instance for the corresponding reduction. 131
3.28 Oscillatory example for FIFO and Random semantics. .......................... 135
4.1 Marge initiates a protocol. ................................................................. 146
4.2 Bart participates in the protocol. ......................................................... 147
4.3 A Twitter screen following all activities. ............................................. 147
4.4 Social Network for scenarios 1 and 2. ................................................. 150
4.5 Social Network for scenario 3. A table belongs to the participant whose column is grey. ................................................................. 150
4.6 (a) $P_1$ and $P_2$ just before operating on the same nodes. (b) CAG discussed in Section 4.3.5 ................................................................. 151
4.7 Status updates generated by scenario 1, as seen by the system. ......... 164
4.8 Social Network for the example in Section 4.4.2. ................................. 174
4.9 Demonstration Network. ................................................................. 177
4.10 Log of the two-way example. ............................................................ 177
4.11 Illustration of the construction in the proof for consistent regret. ....... 178
6.1 A partial execution tree in stratum 1 of the execution tree .................. 189
6.2 A sufficient execution tree with two strata .......................................... 190
6.3 Stratified execution tree for a program with stratified negation. .......... 190
6.4 Experiment 1 datasets. ................................................................. 207
Abstract

This work describes research on novel Web-related data management scenarios. In the first part, we describe the $XPath^L$ language and system, which combines XPath and Datalog by introducing an XPath predicate to the Datalog formalism. In the second part, we introduce the Query Network model, a basic model for social network automation using queries, with its evaluation algorithms. In the third part, we discuss social networks automation using Protocols for Social Networks, which coordinate consistency-preserving (and other) decisions in the network.

We begin by defining the $XPath^L$ data model and language, which combines both XML and relational data. We define the semantics of XPath predicates and provide examples for the usefulness of the model. The concept of values is introduced and investigated. We show some theoretical results about the language. We design evaluation algorithms for recursive and non-recursive queries, and experiment with them using the $XPath^L$ system prototype. In addition, we characterize the complexity of the closely related problem of evaluating conjunctive queries with XPath-inspired predicates on DAGs.

We turn to the Query Network model. We motivate and define the model, which uses queries to abstract the process of proposing and accepting connections in a social network. The model raises the novel database problem of how to evaluate a query whose size is in the order of magnitude of the data being queried. This is in contrast to the traditional database assumption that queries are small and data are large. We design evaluation algorithms for query networks and evaluate them using synthetic graphs as well as real social networks graphs that we derived from the DBLP dataset. We characterize the complexity of several related decision problems.

The third and last chapter is concerned with a future scenario in which social networks participants use protocols in order to efficiently interact in the social network. We introduce a model for a social network in which participants make decisions that are consistent with their friends deci-
sions. The protocols provide participants a means to coordinate the atomic installment of sets of decision changes, which preserve network consistency. We consider both a Twitter-like one-way communication architecture, and a Facebook-like two-way communication architecture. We develop novel methods for controlling the concurrency of multiple protocol instances, which enable increased concurrency, by relaxing the traditional concurrency control isolation requirement.
Chapter 1

Introduction

The fast development of the internet, and particularly of online social networks, has given rise to new data management problems. This research examines three novel data management topics that are closely related to web data management, and to each other. We start with a brief overview of the main topics of the thesis. Then, we outline the results for each of the topics.

1.1 Overview of the Thesis

XPathL - Datalog and Semistructured data. The first topic addresses the joint processing of XML and relational data, which has been a focus of research and development (see, e.g., [52], [26], [57]). We address the case where XML and relational processing is embedded within an application language, which combines the two data types in the same logical framework.

We choose Datalog, which has been proven as a successful formalism for abstracting many query languages, as a basis for XPathL - a new, logic-based object-oriented language for processing relational and XML data. We extend Datalog with a predicate which processes XPath queries on XML documents. By allowing this predicate share variables with classical relational predicates, XPathL provide a framework for uniformly handling relations and XML, that is also extensible to other data types. In addition, we investigate the particular case of non-recursive Datalog queries (Conjunctive Queries).

XPathL is designed as an intermediate language, which is not necessarily exposed to users. The user syntax does not necessarily have to adhere to Datalog conventions, although in some
applications, the language can also serve as the user’s language. The main motivation is to make it easy for language designers and implementors to include advanced querying features in their syntax, and be able to represent them internally using a Datalog-based formalism.

We take a loosely coupled approach for this joint processing of distinct data types. By this, we provide a framework for integrating data types other than XML and relational in query evaluation, which can benefit from the wealth of practical and theoretical knowledge on Datalog.

The $\text{XPath}^L$ data model is defined and researched. We address theoretical questions such as the interplay between objects and values, query evaluation complexity and language expressiveness. We develop query evaluation approaches and methods. We experiment with these methods and show their usefulness using the $\text{XPath}^L$ prototype system, which implements the language in full.

**Query Networks.** We turn to another novel use of the Datalog formalism - in the context of online social networks. Recently, social networks have seen unprecedented popularity and growth. In 2010, Facebook surpassed Google as the most visited site in the US [34], establishing itself as the most popular web application. This popularity has brought large amounts of data to social networks. The Claremont Report on Database Research [6] mentions the big data coming from social networks as a central research challenge.

Nielsen’s report from August 2010 [72] shows that the monthly time spent by internet users on social networks is more than the time spent on e-mail, search, instant messaging, portals and classified ads combined. Increasing the productivity of social networks participants is therefore a pressing challenge.

These developments motivate the novel concept of automatically interacting in the social network. The Query Network is a model which abstracts the automation of the most basic activity in social networks: proposing and accepting connections.

We distinguish between two models. One model only has proposal queries, and it is assumed that every proposal is accepted. The second model also has acceptance queries, which define from whom a participant is willing to accept connections in the social network.

The Query Network model takes a database approach and describes policies for proposing and accepting connections in the social network as queries. We abstract these queries as Conjunctive Queries. The collection of the queries associated with the network participants forms a very large, typically recursive, Datalog program.
In sharp contradiction to the traditional database assumption that queries are small and data are large, this Datalog query is in the order of magnitude of the data being queried. This situation gives rise to theoretical and practical problems that we address in this thesis.

**Protocols for Interaction in Social Networks.** We continue to investigate methods aimed at automating interaction in social networks and increasing participants’ productivity.

In order to make social network automation applications reliable and effective for personal and business contexts, it is necessary to let users safely rely on others’ decisions, automate responses, verify the consistency of decisions with possible user requirements and modify decisions as situations in the network develop.

We propose *protocols* as a means for automated engagement and cooperation for participants in social networks. The inspiration for the use of protocols in social networks comes from many real-life social interactions, in which protocols (that are usually unwritten) are being used.

Protocols for social networks are motivated by the need to alleviate the productivity bottlenecks in social networks. The protocols enable efficient structured communication involving a large number of participants. Participants cooperating using a protocol are not necessarily directly connected to each other (even though messages are sent only between friends). Without structure, such cooperation would be difficult, if not impossible.

Automation with protocols also provides a framework for automated engagement for commercial participants. Applications such as Radian6 [75] and Visible Technologies [95], which offer commercial engagement, have recently been the focus of innovation in the area.

The distributed nature of activities in social networks necessitates that different decision making processes be able to operate concurrently without interfering (and even in synergy) with each other. The methods we present allow multiple protocol instances to operate concurrently on intersecting parts of the social network, as well as help each other succeed in places where isolated protocols would have necessarily failed.

In contrast to concurrency of transactions, the protocol instances are not isolated from each other. This enables the mutual help discussed above. Alleviating the isolation requirement enables increasing the level of concurrency and avoiding abort of protocol instances.
1.2 Datalog and Semi-Structured Data

We define the data model for the $XPath^L$ language, which consists of an XML document and a collection of relational tables. We extend Datalog with $XPath$ expression predicates, of the form $exp[\cdot,\cdot]$. Generally speaking, expression predicates are satisfied by tuples $(x_0,z_0)$ such that XML node $z_0$ is in the result set of the expression $exp$ evaluated at $x_0$ as the context node. Relations and XML documents are integrated by letting relational and expression predicates share variables. We use a least-fixpoint bottom-up Datalog semantics (see [92]) for $XPath^L$ queries (or, programs).

The $XPath^L$ model distinguishes between nodes in the XML document and values of nodes. The value of a node is isomorphic to the tree structure and node labels of the subtree rooted at the node. The relation $Val$ associates between objects and values.

We demonstrate the usefulness of the language with several examples which exhibit the use of nodes, values, recursion, expression predicates and other features of the language. We show that fairly complex data needs can be met with elegant, succinct $XPath^L$ queries.

We investigate several related theoretical problems: The expressiveness of several dialects of the language, the complexity of identifying all the distinct values in a tree and the undecidability of $XPath^L$ satisfiability.

We turn to developing evaluation approaches for $XPath^L$ queries. The Static Tuple Generation approach conservatively populates a relation for every expression predicate which appears in the query. This approach is useful when the same data is frequently queried (for example, an internet site serving many users). The On-Demand Tuple Generation approach populates relations corresponding to expression predicates based on bindings to variables that are found in the course of evaluating the query. This approach is a must for processing large XML documents in a scenario where the preprocessing overhead cannot be shared among many tasks.

We present algorithms for on-demand evaluation of Conjunctive Queries (non-recursive, single-rule $XPath^L$ programs). We present the column-based, join-based and quick evaluation algorithms for conjunctive queries, and compare them in view of several types of queries. For recursive queries, we present the column-based evaluation algorithm, which recursively extends the relations in the course of evaluating rules in the query.

The $XPath^L$ system prototype, which implements the $XPath^L$ language in full, is presented with its components. We experiment with the algorithms, for both conjunctive queries and recursive
Finally, we consider conjunctive queries on DAG-structured data with XPath-inspired predicates. We motivate the problem with scenarios from the Semantic Web and Complex Event Processing. We fully characterize the complexity of evaluating these queries, and compare the results with complexity results for tree-structured data.

1.3 Query Networks

The Query Network is a graph-based model for automated social networks. Query Networks model a near-future scenario in which participants of a social network manage their account using automated tools. In particular, the probable case in which participants propose and accept connections automatically, according to rules, is considered.

A query network is a directed graph, in which every node represents a social network participant, and edges represent connections between participants.

We consider two variants of the model. In the first model variant, every node \( n \) has one query associated with it. The query defines edges of the form \((n, \cdot)\), which are edges such that \( n \) would like to add to the graph (or, friends that \( n \) would like to have). In the second model variant, every node \( n \) has two queries associated with it. The first query has the same semantics as the query in the first model. The second query defines edges of the form \((\cdot, n)\), which are edges such that \( n \) would like to accept a connection, if proposed by a network participant denoted by the first argument.

In the first variant, every edge which satisfies a query is added to the graph. In the second variant, an edge \((n, m)\) is ‘proposed’ by \( n \)’s first query (i.e., satisfies the query), and added to the graph only if ‘accepted’ by \( m \)’s second query (i.e, satisfies the query).

The query language that we use for queries in the query network are Conjunctive Queries (or, CQs). The choice of CQs is motivated by several considerations. CQ is a formalization that has been successfully used in analyzing and abstracting many query languages. CQs have been extensively studied and their practical and theoretical aspects are well understood. In addition, experience shows that the CQ formalism can be successfully extended to handle data types other than relational. This is a crucial characteristic in the dynamic, heterogeneous web environment.

An advanced alternative to conjunctive queries in this context may be SoQL, a new query
language oriented for social networks [80]. We briefly describe SoQL in Section 1.5.

The algorithmic problem that we consider is to produce a fully evaluated network, in which all the possibilities to propose and accept edges have been materialized. For this, we look at queries in the model as a large collection of CQs associated with the network nodes as a Datalog program. We employ a least-fixpoint bottom-up semantics [92] for the Datalog program composed of all the CQs in the network.

From an algorithmic point of view, rule evaluation has to be done on a large scale, an issue which raises many practical and theoretical issues addressed throughout the chapter.

Three algorithms for evaluating very large Datalog queries over a query network of the first variant are presented. The Basic evaluation algorithm is a simple evaluation algorithm, it is used as a baseline for comparison. The Backward-Radius Triggering algorithm reduces the number of query evaluations by triggering only the evaluation of queries whose results could be affected by edges added to the network. The Divide and Conquer evaluation algorithm partitions the network, evaluates each partition separately and merges the results. It has a large potential for parallelism.

For the second variant, four algorithms are presented. The Simple network evaluation algorithm is a naive algorithm used as a baseline for comparison. The Propose-Accept algorithm reduces the number of evaluations of acceptance queries. The Propose-Accept with Backward-Radius algorithm triggers the evaluation of a typically small number of proposal and acceptance queries in each iteration. The Network Partitioning algorithm takes advantage of the clustered nature of social networks.

Implementation of the algorithms is presented. Experimentation with synthetically generated datasets, as well as datasets derived from the DBLP [28] database are presented. DBLP was chosen in order to capture patterns of social activity as reflected in the collaboration between authors of publications recorded in DBLP.

For both models, the results show that among algorithms which do not use graph partitioning, radius-based methods perform better than others. However, the results for both model variants clearly show that methods using graph partitioning (Divide and Conquer and Network Partitioning) are especially efficient. We simulate parallel computation for network partitioning and demonstrate the influence of the number of partitions on performance.

In addition, we also study related theoretical problems. We characterize the complexity of
the following problems: The Network Fixpoint problem, which is to decide whether a network is fully evaluated or not; The Query Substitution problem, which is to decide, for a participant \( u \) whether there exists a proposal query (which can be seen as a social strategy), such that the fully evaluated network will result in the addition of a specific edge. The k-Popularity problem, which is to decide whether there exists a proposal query that \( u \) can use in order to create at least \( k \) new friends; The Adding, Avoiding and Careful Query assignment problems that are related to automatic engagement in social networks, an area of great interest to social networks related problems (see, e.g., [75]).

Extension to the model and variants are proposed.

1.4 Protocols for Social Networks

The first type of protocols that we present are protocols for determination of status regarding entities in the social network. The entities are typically events such as parties, demonstrations, virtual events, commercial offers etc. These protocols coordinate participants’ decisions regarding status determination, which may well depend on status decisions by all or some of their friends.

We define a generic protocol, using three types of messages: initiation, forwarding and feedback. The behavior of this generic protocol is shaped using modifiers, that are simple programming-like building blocks which give the social network participants a variety of useful options for protocols, such as deadlines and forwarding permissions.

The second type of protocols that we define are protocols for changing decisions. We define the notion of consistency in a social network. Each network participant has consistency requirements, which define which combinations of decisions, by the participant and the participant’s friends are acceptable (or, consistent). In a consistent network, all the participants decisions are consistent. A non-coordinated change of status (of one or more participants) may therefore render the network inconsistent. This type of protocol coordinates sets of changes in participants’ decisions so as to ensure that the resulting network remains consistent. We consider two models, one-way and two-way, both for directed networks.

In the (Twitter-like) one-way model, messages can only be sent from a participant to its followers. We present consistency-preserving protocols for changing decisions: for the initiator,
the participant and for the system (which administers the protocol execution). We generalize the protocols for the concurrent case, in which more than one protocol operate on intersecting parts of the network. Not only do the protocols provide concurrency between protocol instances, they also detect opportunities for mutual help. That is, protocol instances that could not have succeeded independently, may succeed by helping each other.

In the (Facebook-like) two-way model, initial messages can be sent from a participant to its followers, but responses for such messages may be sent backwards as well. This allows participants not to expose their decisions to the system, but rather only to their friends, and relieves processing burdens from the system which exist in the one-way model. Here too, we present protocols for coordinated change of status, for a single protocol instance and for concurrent instances.

We implemented proof-of-concept applications for protocols on social networks. For the first type of protocols, we implemented a system and interface which lets Twitter users initiate and participate in such protocols, proving the concept on a real social network. For the second type, we implemented two applications. The first application uses real Twitter accounts, and implements one-way protocols on Twitter. The second application simulates the operation of concurrent two-way protocols without using real social networks accounts.

Finally, we address several related theoretical problems. The intractability results for these problems further reinforce the usefulness of our heuristic protocol approach for coordinating decision changes.

1.5 The SoQL Language

The SoQL (SOCial networks Query Language) is a new language for querying and creating data in social networks. The language is designed to meet the growing need of social networks participants to efficiently manage the large, and quickly growing, amounts of data available to them, as well as automate processes of creating new data. This need is increasingly pressing as social networks gradually become an important working tool for business development and management. SoQL is a step in the direction of meeting the challenges of providing an expressive querying mechanism and automating processes in social networks.

SoQL is an SQL-like language which enables the user to retrieve paths to other participants in
the network, and use a retrieved path in order to attempt to create a connection with the participant at the end of the path. The language can specify complex conditions that a desired path should satisfy. The language also supports retrieving a group of participants which satisfy conditions as a group, and connecting its members to each other. SoQL uses the path and group as data types.

Since SoQL is not yet implemented, we do not present SoQL in the thesis. However, since the publication on SoQL, [80], has been cited 14 times (as of July 2011), we add the paper to the appendix.
Chapter 2

Datalog and Semi-Structured Data

2.1 Introduction

The joint processing of XML and relational data is of major importance in information and knowledge management. Many database management systems support XML data, which is the de facto standard for data representation and exchange over the web, and many websites extensively use relational databases. The introduction of XML processing into applications has been a focus of research and development. There are a number of query languages for processing XML, the best known are XPath [106], XSLT [108], XQuery [107] and SQL/XML [88]. There have also been attempts at embedding XML processing within standard programming languages such as Java in e.g., XJ [52] and JAXB [57], or C#, in e.g., LINQ [65]. Some proposals introduce XML processing and XML schema types into the programming language itself (e.g., XJ [52] and LINQ [65]).

We address the case where XML and relational processing are embedded within an application language. This embedding may be done in various ways. One is JDBC-style [51] or JAXB-style [57] in which calls to well-defined generic methods are embedded within the application. Another is the XJ [52] approach in which the embedded processing is considered as a part of the language and is compiled after performing a source-to-source transformation.

Whichever the approach, it would be useful to have a simple intermediate language that can handle both XML and relational processing. Such a language may also provide a framework for uniformly handling newer kinds of data which may appear in the web (e.g., semantic web data) or distributed data. This intermediate language is not necessarily exposed to users. In general, the
user syntax may be fairly different than that of the intermediate language, although it is conceivable that it also serves as the user’s language in some applications. The main motivation is to make it easy for language designers and implementors to include advanced querying features in their syntax.

The intermediate language we consider is $\textit{XPath}^L$, which is an object oriented extension of classical Datalog [92]. Datalog is a query language for relational databases, in the spirit of logic programming (e.g., Prolog). $\textit{XPath}^L$ operates on a collection of traditional database relations (like Datalog) and an XML document (like XPath). For relations, ordinary Datalog predicates are used whereas for XML data, new types of predicates, the ternary $\textit{XPath}\ \text{predicate}$ and the binary $\textit{Expression}\ \text{predicate}$, are used.

The choice of Datalog for an intermediate language is motivated by the following considerations. First, Datalog rules (essentially conjunctive queries) provide an elegant formalization for many query languages (or important parts thereof) for semi-structured data, relational data (or both) such as XPath [42] (and therefore XQuery), XSLT (which essentially can be mapped to a conjunctive query as it can be mapped to SQL [56]), XCerpt [16], SPARQL [87] (also see [21]) and SQL4X [26]. Second, the idea of new predicate types, such as the XPath predicate, is generalizable to new sorts of data which may become available in the dynamic web environment. Third, the knowledge regarding optimization techniques for Datalog processing (among others, Query-Sub-Query [94] and Magic Sets [13]) may potentially be utilized. This is similar to the "migration" of the non-recursive variant of the Magic Sets technique into SQL [46]. Another example is the usage of Query-Sub-Query as an optimization technique for querying data intensive P2P systems [3]. Lastly, Datalog has a solid theoretical basis and a well understood semantics. This makes it possible to formally specify and analyze properties at the level of this intermediate language.

$\textit{XPath}^L$ takes a loosely coupled approach for integrating XML and relational data. This approach fits well the distribution of data of many sorts and/or processing units over the web. The $\textit{XPath}^L$ runtime is a lightweight application which easily allows factoring in new types of data, with their related processors, and offers Datalog optimization on top. In contrast, $\textit{System RX}$, which extends DB2, [17] takes an evolutionary, tightly coupled approach. We further discuss $\textit{System RX}$ in
In addition, we study the evaluation of stand-alone, non-recursive, Conjunctive Queries (or CQs, to be defined), not necessarily as a part of a Datalog program. An important difference between evaluating Datalog and evaluating a CQ is that a rule in a Datalog program may be evaluated repeatedly, possibly contributing new results on each evaluation. A CQ (not in the context of Datalog) is often evaluated once to obtain the final result.

By allowing predicates of the two types to share variables, we provide a framework for uniformly handling relational and XML data, which is also generalizable to other kinds of data. As we define later on, the use of an XPath expression in a query is done through an expression predicate. Each expression predicate corresponds to an expression relation. Expression relations are, conceptually, built-in relations [92]. Another built-in relation is the unary relation \textit{Root} which contains one tuple with the id of the XML document’s root. Conceptually, our data model is an object-oriented model, in which every object, whether it is an XML node or relational data, is identified by a unique id. Practically, an id of an XML node may be, for example, its offset in the XML document, assigned by the system when the node is first read. Other schemes that support updates are possible. Predicates in a conjunctive query may correspond to relations which contain node ids. These ids are ordinarily inserted to these relations by previous queries. Only ids (of either XML nodes or strings) can substitute for the query variables.

Co-Processing of Relational and XML Data. Our approach to the joint querying over the different types of data is that of co-processing by the XPath processor and the relational processor. We assume that each type of data is given and stored in its original form, and is not completely normalized into a single file type. Keeping the data types in their original forms provides several advantages, as discussed next.

First, there is no need to transform one type of data to the other (e.g., to shred all the XML data). Second, queries posed on data of a certain type need not be rewritten to fit the normalizing type of data. Third, advanced optimization techniques have been developed and are constantly being enhanced for type-specific query processors. We would like to be able to take advantage of these techniques. Fourth, algorithms which follow this approach are significantly easier to generalize to other data types and other query languages. For example, generalization for XQuery can be done by plugging in an XQuery processor, without conceptually changing the evaluation
algorithm. Finally, this approach makes data distribution easier; a site need only support the type of data it stores. Only ids are transferred between the processors.

2.1.1 Chapter Outline

Section 2.2 defines the concepts of the data model that we use, and gives examples of queries. Section 2.3 discusses the static and on-demand evaluation approaches, as well as the evaluation algorithms. Section 2.4 describes the system prototype and shows experimental results. In Section 2.5, theoretical results regarding the expressive power of several language dialects, and equivalence undecidability are proved. Section 2.6 characterizes the complexity of Conjunctive Queries with XPath-inspired predicates over DAGs.

2.2 Data Model

2.2.1 Basic Concepts

2.2.1.1 Datalog

A Datalog program is a set of rules. Each rule has a body, built of atoms, which are predicates with parameters [92], or negated atoms. A parameter can be either a variable or a constant. Each rule has a head which is an atom. The predicates are relation names, either database relations (EDB, Extensional Database) or relations defined by the program (IDB, Intensional Database). \textit{XPath}’s semantics is an extension of the least-fixpoint bottom-up Datalog semantics, namely starting with the EDB and continuing the evaluation by considering both EDB and IDB tuples. Refer to [92] for a formal definition of Datalog semantics, for programs with and without negated atoms.

2.2.1.2 XML and XPath

XML, a standard for representing information, is extensively used in modern databases and on the Web, and has become the standard for representing and exchanging semi-structured data. An XML document can be represented as an ordered tree whose nodes are elements or attributes (of nodes). In many applications XML documents function as the database, either by replacing classical database relations or by accompanying them. Many languages were proposed for
querying XML documents: XPath [106] [104], XSLT [108], XQuery [107] and, in an SQL context, SQL/XML [88] and others. There are two versions of XPath, XPath 1.0 [106] and XPath 2.0 [104]. XPath 1.0 is a simple language for navigating an XML document and selecting nodes from it, processing four data types: Nodeset, Boolean, String and Number [106]. XPath 2.0 is a more powerful language with advanced constructs such as loops. In this text, XPath refers to XPath 1.0.

2.2.1.3 XPath\textsuperscript{L}- Introducing Predicates for XML processing

XPath\textsuperscript{L} introduces into Datalog two new types of predicates dedicated to XML processing, called XML predicates. One type is the binary Expression Predicates and the other is the ternary XPath Predicates. Expression predicates have the form exp[·, ·] and are satisfied by tuples \((x_0, z_0)\) such that node \(z_0\) is in the result set of the expression \(exp\) evaluated at \(x_0\) as the context node (XPath expressions returning String, Boolean or Number are discussed in Section 2.2.5). Expression predicates appear with square brackets, "["", and other predicates - with regular brackets, "]". Therefore, the predicate \(a[x_0, y_0]\) holds if \(y_0\) is an "a"-tagged child of \(x_0\), while \(a(x_0, y_0)\) holds if the tuple \((x_0, y_0)\) satisfies the non-expression Datalog predicate whose name happens to be "a".

The XPath predicate is added in order to allow the bindings of expressions. It is satisfied by triplets \((x_0, e, z_0)\) such that object \(z_0\) is in the result set of the XPath expression \(e\) evaluated at \(x_0\). Note that there are infinitely many expressions which may 'navigate' from \(x_0\) to \(z_0\). Therefore, we restrict \(e\) to be a member of a finite set of "allowed" expressions, discussed in Section 2.2.4.2.

XML predicates appear as predicates in rule bodies only. Their parameters, however, may appear in both the head and the body of rules. A relational predicate name \(p\) starts with a lowercase letter, and corresponds to a relation whose name is the same as \(p\)'s name, but with the first letter capitalized.

Safety and Stratification of Negation. We restrict attention to XPath\textsuperscript{L} programs that are safe. We require that negation is stratified, i.e., if a predicate \(q\) is defined by negating another predicate \(p\), then \(p\) is not defined, directly or indirectly, using \(q\). Refer to [92] for formal definitions of safety and stratification.
2.2.1.4 Values

An XML document can be modeled as an ordered labeled tree, called an XML data tree. Two distinct nodes in a document have a different identity even if they represent identical document fragments, ignoring order of attributes. We say that such nodes have the same value. Formally, the value of a node \( v \) is the root of a unique XML data tree, disjoint from the database tree, which is isomorphic to the tree rooted at \( v \). Classical Datalog is value oriented. XML processing, however, requires considering both the concept of a node’s identity (id) and the concept of a node’s value. XPath\(_L\) is object oriented, basically manipulating object ids. Therefore, values are also objects, but of a special kind. A value object has no parent and its descendants in the tree, if any, are non-value objects. Section 2.2.2 includes formal definitions.

The XPath\(_L\) data model includes the binary built-in relation Val. A tuple \((a,b)\) is in Val if \( a \)'s value is \( b \). Using value-checking predicates, querying values is allowed. Values can also be stored in relations. Regarding values as objects (as opposed to, say, the XML fragments themselves with which they have a one-to-one correspondence) allows us to have a uniform object model. This in turn makes programs simpler and they do not have to carry around, and process, long strings. Values are formally defined as a part of the data model in Section 2.2.2.5 and are further discussed in Section 2.2.3.4.

2.2.2 Object Oriented Data Model

2.2.2.1 Objects

An object \( o \) is an entity that is uniquely identified by its object identifier (id), denoted \( id(o) \), which is a string. An object is either a string object or a node object. A node object is either a text node object (TNO), an element node object (ENO) or an attribute node object (ANO).

We use TNOs to represent text nodes in XML documents. Each TNO \( t \) is associated with a string, \( text(t) \). Many TNOs may be associated with the same string. Let \( \{o_s|s \text{ is a string}\} \) be the set of string objects. The function \( val(\cdot) \) associates each TNO \( t \) with the string object corresponding to \( text(t) \), namely \( val(t) = o_{text(t)} \). We use ENOs to represent element nodes in XML documents or XML fragments (except for fragments which consist only of a text node). Each ENO \( e \) is associated with a label, \( label(e) \), which is a string. We use ANOs to represent attribute nodes. Each ANO \( a \)
is associated with a name, \( \text{name}(a) \), and with a string, \( \text{text}(a) \).

In addition, an ENO \( e \) is associated with a (possibly empty) ordered list, \( L_e \), and with a (possibly empty) set, \( A_e \). \( L_e \) consists of child node objects. A child node object may be either an ENO or a TNO. \( A_e \) consists of ANOs with unique names.

### 2.2.2.2 XML Trees

An XML document, say \( d \), is represented by an ENO whose label is the empty string, which represents the document root element. Its only child is the ENO representing the document element ("uppermost" element). An XML fragment is represented directly using the ENO representing its root, whose label has to be a non-empty string. Each element \( n \) in a document \( d \) or in a fragment \( f \) is represented by an ENO \( n_e \) s.t. \( n_e \)'s label (as defined for XML elements) is the string \( \text{label}(n_e) \). Each text node \( b \) in \( d \) or \( f \) is represented by a TNO \( b_t \) s.t. \( b_t \)'s text (as defined for XML elements) is the string \( \text{text}(b_t) \). Each attribute \( a \) in \( d \) or \( f \) is represented by an ANO \( a \) s.t. \( \text{name}(a) \) is the attribute’s name and \( \text{text}(a) \) is its value. Children of an element represented by an ENO \( e \) are represented by the ENOs or TNOs in the ordered list \( L_e \), and attributes - by the elements in the set \( A_e \). We call this structure an XML tree.

### 2.2.2.3 Relations

Being an extension to the relational data model, our data model also consists of relations. Tuples may contain Strings, Boolean, Numbers and object ids.

### 2.2.2.4 Isomorphism of Nodes

Intuitively, two ENO objects are isomorphic if they represent XML documents or fragments with the same tree topology, labels, text node values, attributes and attributes’ values.

Two TNOs \( v_1 \) and \( v_2 \) are isomorphic if \( \text{val}(v_1) = \text{val}(v_2) \), i.e., \( \text{text}(v_1) = \text{text}(v_2) \). Two ANOs \( a_1 \) and \( a_2 \) are isomorphic if \( \text{name}(a_1) = \text{name}(a_2) \) and \( \text{text}(a_1) = \text{text}(a_2) \).

Two ENO objects \( v_1 \) and \( v_2 \) are isomorphic if:

- \( \text{label}(v_1) = \text{label}(v_2) \), and
- \( |L_{v_1}| = |L_{v_2}| \) (i.e., the lists have the same number of objects), and for \( 1 \leq i \leq |L_{v_1}| \) the \( i \)'th element in \( L_{v_1} \) is isomorphic to the \( i \)'th element in \( L_{v_2} \) (list elements can be either ENOs or TNOs).
and

- \(|A_{v_1}| = |A_{v_2}|\) and every element in \(A_{v_1}\) has an isomorphic element in \(A_{v_2}\) and vice versa.

Next, we use node isomorphism to define value objects.

### 2.2.2.5 Value Objects

Value objects are a proper subset of all objects:

1. All string objects are value objects.

2. All non-empty-string labeled ENO, an ANO or a TNO \(w\) s.t. for all ENO nodes \(u\), \(w\) is not a child of \(u\), is called a value object. Not having a parent implies that \(w\) is not a part of a document. However, \(w\) has a unique id. We postulate that for every ANO, TNO or ENO \(u\), there exists a unique value object which is isomorphic to \(u\).

We extend the function \(val\) defined above to map each object \(v\) to that unique object \(w\), i.e., \(val(v) = w\). In particular, this implies that for a value object \(w\), \(val(w) = w\). Note that \(val\) as defined above for string objects is thus naturally extended. In other words, for each equivalence class of isomorphic sub-trees, there is a unique value object that is isomorphic to all the class members and is not part of a "larger" tree. More than one node object may be associated with the same value object, but each node is associated with exactly one value object. In Section 2.2.4.1, we define the binary relation \(Val\) discussed above, whose tuples are \((id(u), id(w))\) where \(val(u) = w\). In rules, we use the corresponding binary predicate \(val\). Note that proper descendants of a value object \(w\) are not value objects, as they each have a parent.

### 2.2.3 \(XPath^L\) Examples

In this section, we present examples of \(XPath^L\) query programs in order to highlight the language’s main features.

#### 2.2.3.1 Same Item Categories Example

The XML document for this example is illustrated in Figure 2.1. The "items" element is an ordered sequence of the items most frequently purchased by a customer. For example, the item most
frequently purchased by Rachel is *skim milk*. Relation *C* relates items (first column) and their category (second column). Some of *C*’s tuples are illustrated in Figure 2.1. For the sake of readability, in the table of Figure 2.1, we display actual strings and not ids of string objects. The actual relation *C* consists only of ids.

The knowledge as to which customers have similar behavior is valuable. Therefore, we would like to retrieve pairs of customers whose *k* most frequently purchased items belong to the same category. That is, customers *c*₁,*c*₂ such that for every *i* ≤ *k*, the *i*’th most frequently purchased item by *c*₁ belongs to the same category as the *i*’th most frequently purchased item by *c*₂.

The following *XPath* program retrieves such pairs ("fol-sib" abbreviates "following-sibling"). An explanation follows the code.

1. q(X,Y,0)←self::customer/fol-sib::customer[X,Y].
2. q(X,Y,L)←q(X,Y,K),L=K+1,it(X,I,L),it(Y,J,L),
3. val(I,I’),val(J,J’),c(J’,M),c(I’,M).
4. it(X,Y,1)←root(W),//customer[W,X],//item[1]/text()[X,Y].
5. it(X,Z,L)←it(X,Y,K),L=K+1,../fol-sib::item[1]/text()[Y,Z].

Figure 2.1: Supermarket purchases XML document. Text nodes are in *italics*.
The rules in lines 4-5 define the predicate \( it \). Predicate \( it \) is satisfied by 3-tuples \((x, y, n)\) s.t. \( y \) is the id of the text node of the \( n \)’th ”item” element under the ”items” element of the customer element whose id is \( x \). The rule in line 4 inserts into \( it \) tuples for the first ”item” element, and the rule in line 5 adds a tuple for its immediate sibling of ”item” elements already existing in \( it \). The id of the text node of the already existing ”item” is bound to \( Y \), and the id of the text node of the immediate sibling - to \( Z \). The rules in lines 1-3 define the predicate \( q \) which is satisfied by 3-tuples \((a, b, k)\) s.t. for every \( i \leq k \), the \( i \)’th most frequently purchased item by customers with ids \( a \) and \( b \) belong to the same category. Every two distinct customer elements satisfy the rule in line 1, since it requires zero category matchings. The rule in lines 2-3 requires that there are \( K \) matchings, and that the \( K+1 \)st ”item” text nodes (whose ids are bound to \( I \) and \( J \)) have strings (bound to \( I' \) and \( J' \)) which appear in the relation \( C \) with the same category (bound to \( M \)). The use of \( val \) is needed because \( I \) and \( J \) are bound to ids of nodes, and therefore cannot match the contents of the relation \( C \). \( I' \) and \( J' \), on the other hand, are bound to string object ids, and can therefore match the contents of \( C \).

Querying \( q \) provides the pairs of customers for a specific value of \( K \). The following rule populates the relation \( B \) with pairs of customers with three matchings.

\[
1 \ b (X, Y) \leftarrow q (X, Y, 3).
\]

For example, according to the data in Figure 2.1 the ids of the ”customer” elements corresponding to Jacob and Rachel are not in \( B \), but they do belong to \( B' \).

\[
1 \ b' (X, Y) \leftarrow q (X, Y, 2).
\]

### 2.2.3.2 Top Selling Items Example

In the previous example, the use of the predicate \( val \) was limited to ”translating” from TNOs to string objects. Next, we present an example in which \( val \) is used over complex subtrees. Suppose that the supermarket has another XML document with another view on the popularity of items. In this document, whose form is illustrated in Figure 2.2, each ”year” element consists of ordered
sequences of the three best selling items in the different weeks of that year. For example, in the first year, in the first week, the best selling item was *skim milk*, the second best selling item was *soap bar* etc. We would like to know if there are two years s.t. in each of them, there is a list of *k* weeks, in the same order (but not necessarily consecutive), such that the *i*’th (1 ≤ *i* ≤ *k*) week in both lists has exactly the same (ordered) set of three best selling items. This means that each one of the *k* elements with tag "top" in one year has to have the same value as its counterpart in the other year.

The following *XPath* program solves the problem.

1. \( q(X,Y,B,D,1) \leftarrow \text{self::year/fol-sib::year}[X,Y] \),
2. \( \text{week/top}[X,B], \text{week/top}[Y,D], \text{val}(B,M), \text{val}(D,M) \).  
3. \( q(X,Y,P,Q,L) \leftarrow L=K+1, q(X,Y,B,D,K) \),
4. \( \ldots /\text{fol-sib::week/top}[B,P], \text{val}(P,M) \),
5. \( \ldots /\text{fol-sib::week/top}[D,Q], \text{val}(Q,M) \).

The rule in lines 1-2 inserts into \( q \) the ids of two year elements, the ids of two isomorphic "top" elements and the constant 1. Isomorphism is checked using the predicate \( \text{val} \). The rule in lines 3-5 checks for "top" elements which follow "top" elements already existing in \( q \). They are required to
have the same value (i.e., to be isomorphic). If found, they are inserted into $q$ with an incremented counter. Here too, the relation $Q$ can be queried for a specific $k$. The use of $val$ eliminates the need of writing of a cumbersome query for checking the isomorphism of the subtrees. Moreover, using $val$ allows the same query to be used even if the file is changed to include, say, four items in each “top” element rather than three. One can notice that while the program is very succinct, a naive implementation is not efficient. This motivates work on optimizing programs.

### 2.2.3.3 Basketball Example

The XML document on which the program operates is a log of basketball games played by a team. The document’s form is illustrated in Figure 2.3. Each of the direct children of the root, labeled “game”, is the recording of the different positions taken by the team players during a game. The positions appear with start and end times. For example, in the one game presented in full in Figure 2.3, between 06:34 and 09:15, Johnson was at the point guard position, Jordan - at the second guard, etc.

For analysis, we are interested in retrieving games in which the first $k$ “positions” configurations are the same, where $k$ is a parameter. The following XPath program implements this by building the ternary relation $Res$. After evaluation, a tuple $(game_1, game_2, k)$ is in $Res$ iff the first $k$ “positions” configurations for $game_1$ and $game_2$ are identical. The following code defines $Res$. 

![Figure 2.3: Basketball games XML document. Text nodes are in italics.](image)
Note that \( Oc \), an IDB predicate used in the program, is defined in the second code fragment which follows an explanation about \( Res \). The two fragments together form the whole program.

\[
1. res(X, Y, 0) \leftarrow \text{fol-sib::game}[X, Y].
2. res(X, Y, T) \leftarrow T=K+1, res(X, Y, K), oc(X, Q, T), oc(Y, B, T), positions[Q, Q'], positions[B, B'], val(Q', M), val(B', M).
\]

Every two distinct "game" elements satisfy the rule in line 1, since the zero first positions configuration are (vacuously) identical for every two games. The recursive rule in lines 2-3 uses the ternary relation \( Oc \) defined below, which contains a tuple \((a, b, n)\) iff \( b \) is the \( n \)-th "onCourt" element of \( a \), a "game" element. The body requires that the \( K+1 \)-st "positions" element in the game bound to \( X \), namely \( Q' \) and the \( K+1 \)-st "positions" element in the game bound to \( Y \), namely \( B' \), have the same value (i.e., represent the same document fragment, and in our case, that the same positions are manned by the same players). The rules for \( Oc \) are shown below, followed by an explanation.

\[
1. oc(X, Y, 1) \leftarrow \text{self::game/onCourt}[1][X, Y].
2. oc(X, Z, T) \leftarrow T=L+1, oc(X, Y, L), fol-sib::*[1][Y, Z].
\]

The rule in line 1 inserts into \( Oc \) all the games with their first "onCourt" elements, and the third argument is therefore "1". The rule in line 2 adds to \( Oc \) every immediate sibling of an "onCourt" element already in \( Oc \) in this case (with an incremented third argument).

Querying \( Res \) provides the games for a specific value of \( k \). For example, the following rule populates the relation \( A \) with games whose first seven positions configurations are identical:

\[
1. a(X, Y) \leftarrow res(X, Y, 7).
\]

The program demonstrates the use of two important features in \( XPath^L \). The first is recursion, a basic Datalog feature; the \( res \) and \( oc \) predicates are defined in terms of themselves. The definition of \( res \) also uses the predicate \( val \), which allows one to refer to the \textit{value} that a node represents, and not to its id. Using \( val \), we can identify that positions in game intervals were manned by the same players without having to specify in the query the many details of the structure of "positions". Also, we can use the same query for documents in which the element "positions" is structured differently, as long as the requirement is that they have the same value.

**Querying Values.** Suppose that in addition to requiring that positions elements have the same value, we would also like to add to the condition that Bird follows Jordan in the positions element. Then, we add (as line 4) in the program fragment which defines \( res \) the following expression
predicate:

descendant::*[text()='Jordan']/
following::*[text()='Bird'][M,D].

Note that when querying a value, results are influenced by the scope of the query being disjoint from the entire document. For example, the added expression predicate evaluated on a binding for $B'$ may yield a result which includes nodes from other positions elements, since following::* would not be processed in the context of the value (but rather in the context of the whole tree).

2.2.3.4 More On Values

The concept of values has several advantages. First, it allows the isolation of subtrees that appear several times in the XML data, similar to projection in relational databases. This isolation, in turn, allows for clearer programs. If one would like to query such repeating subtrees, one can simply query the value viewed as a tree and not worry about possible side-effects had the query been posed in the context of the original data. Such side effects may include navigating to unexpected portions of the data tree. Once the concept of a value is part of the language, and an efficient implementation of the concept is provided, querying can be more efficient. For example, if one would like to extract from a set $S$ of subtrees elements satisfying a property $p$ (which is local to the subtree), one could, in principle, pose a query against all the elements in $S$. However, it suffices to pose the query to check $p$ on possibly few values, representing the equivalence classes of isomorphic sub-trees, corresponding to the elements in $S$. It is unlikely that a query processor will perform such an optimization on its own since it needs to determine that the query will give the "same" result on identical sub-trees that are located at different parts of the data tree.

**Querying Values.** Suppose that in the example in Section 2.2.3.2 we also require that the best selling item in the "top" ENOs bound to $P$ and $Q$ is skim milk and that one of the other two is oranges. Then, we can add to each of the lines 2 and 5 the following predicate:

```
item[text()="skim milk"]
/following::*item[text()="oranges"][M,W]
```

As noted before, since we query a value, the result is influenced by the scope of the query not being the entire tree. For example, evaluating this expression on a binding to $P$ must also consider
"item" nodes from other "top" elements, since following::item would not be processed in the context of the value (but rather in the context of the whole tree).

**Value Containment.** Let \( v_1 \) and \( v_2 \) be value objects. We say that \( v_1 \) is *loosely contained* by \( v_2 \) (and that the predicate \( lValCon(v_1, v_2) \) holds) if it is possible to remove nodes and edges from \( v_2 \) s.t. the value of the resulting object is \( v_1 \). We say that \( v_1 \) is *exactly contained* by \( v_2 \) if there exists a node in \( v_2 \) whose value is \( v_1 \) (and that the predicate \( eValCon(v_1, v_2) \) holds).

For example, consider \( u_1 \) and \( u_2 \), two "items" elements in a document structured similarly to the one in Figure 2.1. If for each of \( u_1 \)'s "item" children there exists an "item" element with the same value in \( u_2 \) and in the same order, we say that \( u_1 \)'s value is loosely contained in \( u_2 \)'s value. Note that \( u_2 \) may have other children as well. According to the definition of exact containment, if \( u_1 \) has an "item" child element, \( u_3 \), which has an isomorphic element in \( u_2 \)'s tree, we say that \( u_3 \)'s value is exactly contained in \( u_2 \)'s value.

A simple implementation of a mechanism for determining whether a predicate \( lValCon(v_1, v_2) \) holds may use an XPath processor. First, an XPath expression is created from \( v_1 \) by stating the structure of the ordered tree using "child" and "following-sibling" axes. For example, the expression corresponding to the leftmost "top" element in Figure 2.2 is:

\[
\text{desc-or-self::top[child::item[text()="skim milk"]]/fol-sib::item[text()="soap bar"]/fol-sib::item[text()="oranges"]}
\]

Then, we require that the expression is satisfied when evaluated with \( v_2 \) as context (i.e., on the root of the XML data tree corresponding to \( v_2 \)). A solution in this spirit for *exact containment* can be implemented using an XPath expression which requires an exact match.

**Value Similarity.** Another useful predicate is one which checks for *similarity* of values. \( \text{valSim}(v_1, v_2, t) \) holds if the similarity between \( v_1 \) and \( v_2 \), according to some similarity measure, exceeds a threshold \( t \). Candidates for the similarity measure are, e.g., the tree edit distance or element simulation distance [73]. Note that while value containment necessarily considers order between children in a tree, value similarity allows relaxing this requirement. For example, the

---

1Loose containment between values cannot be expressed using the predicate \( \text{val} \) which requires *exactly* the same structure. On the other hand, \( eValCon(v_1, v_2) \) is equivalent to \( \text{//}[*[v_2, X], \text{val}(X, v_1)] \) (where \( X \) is a new variable). Note that this feature need not be implemented using this simple macro expansion.
previous example could be modified to look for similarity between shopping habits which does 
not necessarily consider the order between "item" elements. The precise semantics of \textit{valSim} is 
application dependant and is therefore considered as an externally defined predicate.

2.2.4 The Database

\textit{XPath} programs are evaluated with respect to a database. The database consists of a collection 
of traditional relations and an XML document, \(d\), whose representing tree’s root \(r\) is in the unary 
relation \textit{Root} defined below.

The domain of entities, \(\text{Dom}\), that can populate tuples (and be bound to variables) consists of \texttt{id}s
of:

1. String objects.
2. ANOs, TNOs or ENOs in the tree rooted at \(r\).
3. Value objects associated with the objects in 2 and with the descendants of these value objects 
(since proper descendants of value objects are not value objects).

Intuitively, tuples in the database relations may ”contain” either references to strings or refer-
ences to nodes in trees. We distinguish between the EDB and the IDB [92].

2.2.4.1 EDB Relations

In what follows, we list the EDB relations in the database. Object \texttt{id}s appearing in EDB tuples are 
restricted to belong to \(\text{Dom}\), the domain of entities defined above.

\textbf{Root}. \textit{Root} is a unary relation consisting of a single tuple; the column entry is the id of the 
ENO representing the root of the document’s XML tree.

\textbf{Value Related Relations}. \textit{Val}, \textit{LValCon}, \textit{EValCon} and \textit{ValSim} are relations corresponding to 
the predicates val, \textit{IValCon}, \textit{eValCon} and \textit{valSim}, respectively.

\textbf{Expression Relations}. For every XPath query there exists a relation whose tuples satisfy the 
corresponding expression predicate. For example, the expression \texttt{a/*/self::d} and the expression 
relation \texttt{a/*/self::d[., .]} (note that the expression need not belong to the set \(E_d\) defined below).
**Xpath.** *Xpath* is a ternary built-in relation corresponding to a predicate used mainly for binding XPath expressions to variables. The relation *Xpath* contains the set of all tuples \((idc, exp, idr)\) s.t.: 

1. \(idc\) and \(idr\) are ids of node objects (intuitively, context and result nodes).
2. \(exp\) is the id of a string object or a TNO object (intuitively, the expression).
3. \(val(exp)\) is the id of a string object whose corresponding string is in the set \(E_d\) defined below, where \(d\) is the XML tree or fragment corresponding to the tree to which \(idc\) belongs.
4. The result set on the XML document or fragment to which \(idc\) belongs, of the XPath expression whose string object id is \(val(exp)\), with the XML node whose id is \(idc\) as the context node, contains the XML node whose id is \(idr\).

Intuitively, \(E_d\) is a finite set of "allowed" XPath expressions in the context of the tree \(d\), and is defined formally as follows. \(E_d\) is necessary in order to ensure navigation safety as explained next.

### 2.2.4.2 The set \(E_d\) of Allowed Expressions

We would like to allow the binding of expressions to variables, i.e., bind expressions to variables in the second argument of an XPath predicate. However, there are infinitely many expressions which navigate between any two given nodes. For example, in many trees, if \(a/b\) navigates between nodes \(x\) and \(y\), so does \(a/b/child::*/parent::*\). We therefore define a finite set \(E_d\) designed to capture every possibility for an expression with respect to (w.r.t.) \(d\), a given XML document or fragment. Consider \(A\), a \(|d| \times |d|\) matrix, where \(|d|\) is the number of nodes in the tree corresponding to \(d\). Each column, say \(i\), represents a different node in \(d\)'s tree, \(n_i\), and so does every row. Values of entries in the matrix are either true or false. An XPath expression \(e\) satisfies a matrix \(A\) if for all pairs \(i, j\), if the entry \(A_{i, j}\) is true (respectively, false) then \(e\) evaluated at node \(n_i\) as context yields (respectively, does not yield) node \(n_j\) in the result set. For a matrix \(A\), let \(e_A\) be an expression s.t. \(e_A\) satisfies \(A\) and \(e_A\) is the one expression built as follows.

For every true entry in the matrix \(A\), say \(A_{i, j}\), one disjunct \(e_{i, j}\) appears in \(e\). \(e_{i, j}\) is an expression which navigates from \(n_i\) to the result set \(\{n_j\}\). \(e_{i, j}\) is: (1) a sequence of ”parent::*” steps whose result set is the document’s root, followed by (2) a sequence of ”child::*[k]” steps, which
navigates along the path from the root to \( n_j \). In each step, \( k \) is the relative position of a node among its siblings. If \( n_i \) and \( n_j \) are the root, \( e_{i,j} = \text{self::*} \). For example, consider a tree in which there is one ”a”-tagged element with two ”b”-tagged children elements (and no other elements). If \( n_i \) is the first ”b”-tagged node in this tree and \( n_j \) is the second, then \( e_{i,j} = \text{../*}[2] \). The whole \( e_A \) is the disjunction of \( e_{i,j} \) s.t. \( A_{i,j} = \text{true} \). In \( e_A \), \( e_{i,j} \) appears in the disjunction before \( e_{i,k} \) iff \( n_j \) precedes \( n_k \) in the document order.

\[
E_d = \{ e_A \mid A \text{ is a } |d| \times |d| \text{ Boolean matrix} \}. \] We call these expressions the allowed expressions. Note that w.r.t. \( d \), every XPath expression has an equivalent expression in \( E_d \), as the result of each expression can be captured with a matrix. Note that this is a semantic definition, and not a design for implementation.

### 2.2.5 XPath Expressions Returning String, Boolean or Number

Up to this point, we discussed only XPath expressions which return node-sets. XPath, however, has three more types of return values: String, Boolean and Number.

The model can be extended to support such queries, by adding (in addition to the objects already existing in the model) the following value objects:

- Two value objects, one corresponding to the Boolean value \texttt{true} and the other corresponding to the Boolean value \texttt{false}. As usual, only the ids of these objects are recorded in relations. Given an XPath expression \( exp_b \) whose return value is Boolean, the 2-tuple \((x_0, b_0)\) satisfies the binary predicate \( exp_b[\cdot, \cdot] \) if \( exp_b \), evaluated at a context node whose id is \( x_0 \), yields the Boolean value whose corresponding value’s object id is \( b_0 \).

- We extend the model with a set of number objects. The set is \( \{o_n | n \text{ is a number}\} \). A number represents a floating-point number that can have any double-precision 64-bit format (IEEE 754 value) [106], which includes a special ”Not-a-Number” value (NaN), positive and negative infinity, and positive and negative zero. Relations record only the ids of objects. Given an XPath expression \( exp_n \) whose return value is a number, the 2-tuple \((x_0, n_0)\) satisfies the binary predicate \( exp_n[\cdot, \cdot] \) if \( exp_n \), evaluated at a context node whose id is \( x_0 \), yields the number whose corresponding value’s object id is \( n_0 \).
String objects already exist in the model as defined above. Given an XPath expression \( exp_s \) whose return value is a string, the 2-tuple \((x_0, s_0)\) satisfies the binary predicate \( exp_s[\cdot, \cdot] \) if \( exp_n \), evaluated at a context node whose id is \( x_0 \), yields the string whose corresponding value’s object id is \( s_0 \).

Note that not only tuples in expression relations can contain ids of the new types. Other relations may contain them as well.

### 2.3 System, Evaluation Approaches and Algorithms

We present evaluation approaches and algorithms for \( XPath^L \) programs and for \( XPath^L \) conjunctive queries, which are non-recursive \( XPath^L \) programs that consist of a single rule. We restrict attention to \( XPath^L \) programs with relational predicates and expression predicates only (i.e., without the ternary predicate \( xpath \) and without the predicates \( lValCon, eValCon \) and \( valSim \)).

#### 2.3.1 Two Evaluation Approaches

Access to XML data is done using expression predicates and the \( Val \)-related predicates. An important issue is when and how to determine that a ground instance of such a predicate represents a tuple which belongs to the corresponding relation. The question is when, how and to what extent should these relations be populated. Two evaluation approaches are considered, \( Static \) and \( On-Demand \).

**Static Tuple Generation.** The Static approach conservatively populates a relation for every expression predicate which appears in the program with all the pairs of nodes from the tree (as well as corresponding values) that satisfy the expression predicate. This approach may populate the expression relations with tuples that are not necessarily used in the evaluation of the query. This may result in very large relations. Also, evaluating every XPath expression in the program at every node in the tree in order to populate a relation (even if the result is not a large relation) is costly. In a scenario where a single query is evaluated, such preprocessing renders this approach non-scalable. However, in scenarios where the same data is frequently queried (for example, an internet site which provides a large number of users different views of its data, using queries that combine the same or similar expressions as ”building blocks”), the preprocessing overhead is
shared among many tasks. The static approach works similarly for conjunctive queries (single rule
\textit{XPath}^L) and for recursive queries. Techniques to expedite the creation of built-in relations for the
static approach are discussed in Section 2.3.2.

\textbf{On-Demand Tuple Generation.} Static tuple generation blindly generates all the possibilities
for satisfying an expression predicate \textit{exp}[\cdot, \cdot], or \textit{val}(\cdot, \cdot) without using the information about
bindings of variables to nodes which may be available at run-time by evaluating conjuncts other
than \textit{exp}[\cdot, \cdot]. The \textit{On-Demand} approach uses this information in order to substantially reduce the
number of XPath evaluations and the cardinality of the built-in relations. This approach is a must
for processing large XML documents in a scenario where the preprocessing overhead cannot be
shared among many tasks. Algorithms taking the On-Demand approach should consider whether
the query is recursive or not. Later on, we present On-Demand algorithms for conjunctive queries
and for multi-rule, typically recursive \textit{XPath}^L queries.

2.3.2 The Static Approach

The static approach has a large preprocessing overhead. Techniques to expedite the complete cre-
ation of the built-in relations were devised and implemented.

\textbf{The Relation} \textit{Val}. Checking the isomorphism of two nodes is linear in the number of nodes in
the smaller of the XML sub-trees rooted by the nodes. We would like to avoid as much as possible
the checking of isomorphism. We use hash codes in order to obtain a ”fingerprint” of the value
of a node. Then, only nodes whose hash codes are equal are checked for isomorphism. These hash
codes are also useful for the On-Demand approach. Naive assignment of hash codes requires reading
and hashing the whole subtree rooted at a node. Instead, we build the hash codes incrementally
as follows.

If a node \(v\) has no children, its hash code \(q(v)\) is the hash code given by the Java method
\textit{String.hashCode()} [55] (abbreviated here as \(h())\) to its tag (for an element) or text value (for a text
node). If \(t\) is \(v\)’s tag, then \(q(v) = h(t)\).

If a node \(v\) has children \(u_1, u_2, \ldots, u_n\), where \(t\) is \(v\)’s tag and ”+” denotes string concatenation, then
\(q(v) = h(\ldots h(h(h(t+q(u_1))+q(u_2))+q(u_3))\ldots+q(u_n))\). In other words, \(v\)’s tag is concatenated
to $q(u_1)$ and $h()$ is operated on the result, which is concatenated to $q(u_2)$. Function $h()$ is again applied to the result, and so on up to and including $u_n$. The procedure requires limited memory space for computing a single hash code. We populate $Val$ using these hash codes as follows. The XML document is traversed, and every node visited is put into a hash table so that nodes with the same hash code are in the same bucket. After the traversal, only nodes in the same bucket are checked for isomorphism. Ids of isomorphic nodes are inserted into $Val$ with another id which uniquely represents their value. Let $n$ be a node associated with value node $v$. Node $v$ is also the value node associated with (non-value) descendants nodes $w$ of value nodes such that $n$ is isomorphic to $w$. A tuple $(w,v)$ for each such possibility is also inserted to $Val$. Note that the creation of $Val$ is done once for a given XML file. If the same document is queried by different queries, $Val$ needs only be populated once. Experiments with populating $Val$ are reported in Section 2.4.4.

**Expression Relations.** In the static approach, we create and fully populate the expression relations for all the expressions which appear in the query. Let $Exp$ be an expression relation, corresponding to an expression $exp$ which returns a node set. We populate $Exp$ with all the 2-tuples of nodes $(x_0,y_0)$ such that $y_0$ is in the result set of the expression $exp$, evaluated at the context node $x_0$. As for relations corresponding to expressions returning Boolean, Number or String, these are populated with 2-tuples containing the context node and the value object corresponding to their return value. A naive implementation would evaluate the expression $exp$ at every node in the XML document. However, with an XPath processor, we obtain cheaply the set of nodes in the first column of $exp$ by evaluating the XPath query $//*[exp]$ at the document’s root (this is semantically different than evaluating $exp$). Then, we evaluate $exp$ at a usually small number of nodes and populate the XPath relation $Exp$ with the outcome. When a new query is evaluated, more computations are necessary if the query has new expression.

### 2.3.3 On-Demand for Conjunctive Queries

We present algorithms for the evaluation of $XPath^L$ Conjunctive Queries, with negation, in which rule bodies have relational predicates and expression predicates.
2.3.3.1 Column-Based On-Demand

Consider the following conjunctive query:
\[ h(V) \leftarrow q_1(U_1), q_2(U_2), \ldots, q_n(U_n), \]
\[ \text{exp}_1[W_{11}, W_{12}], \text{exp}_2[W_{21}, W_{22}], \ldots, \text{exp}_m[W_{m1}, W_{m2}] \].

\( \bar{U}_i \) are sequences of variables and constants, and \( W_{ij} \) are variables. We assume that the constants in the query do not refer to XML elements, since such constants would have to be ids of nodes, which are very unlikely to appear in a query. The \( q_i \) are relational predicates, and are marked as treated. The rest of the predicates are marked as untreated. At this point, we restrict attention to queries without negated predicates. Negation is considered later on.

**Algorithm Sketch.** The pseudo-code for Basic Column-Based On-Demand appears in Algorithm 1. The Column-Based On-Demand evaluation algorithm populates expression relations by considering known supersets of bindings to their first argument. We call a variable \( X \) distinguished at a certain point in time if \( X \) is a first variable in at least one untreated expression predicate and \( X \) appears as an argument in a treated predicate. The main loop is repeated until all predicates are treated. In the main loop, for every new distinguished variable \( X \) we build a unary relation \( B_X \). \( B_X \) is called \( X \)’s binding column, and it contains a superset of the possible binding of nodes to \( X \). \( B_X \) is assigned the intersection of single column relations which are all the projections on attribute \( X \) from all the treated relations having \( X \) in their schema. Observe that elements that are not in this intersection cannot be a part of a satisfying substitution for \( X \). Therefore, expression relations whose first argument is the variable \( X \) need not be populated with tuples whose first element is not in \( B_X \).

Expression predicates whose first argument has a binding column are populated by evaluating the XPath expression of the predicate with the elements in the binding column serving as context nodes. After an expression relation \( e[X, Y] \) is populated, it is marked as treated.

If at any point there are no distinguished variables but there are still untreated predicates, an arbitrary untreated expression relation \( e_0[Z, W] \) is chosen (we call such a predicate an orphan). A binding column \( B_Z \) containing all the nodes in the XML document is created, and the algorithm continues.
Input:
An XPath\textsuperscript{L} rule (conjunctive query):
\[ h(\overline{V}) \leftarrow q_1(\overline{U}_1), q_2(\overline{U}_2), \ldots, q_n(\overline{U}_n), \]
\[ \text{exp}_1[W_{11}, W_{12}], \text{exp}_2[W_{21}, W_{22}], \ldots, \text{exp}_m[W_{m1}, W_{m2}] .\]

Output:
A \( |V| \)-ary relation \( H \) with tuples that represent the result of the query.

Method:

1: \textbf{while} there exists an untreated predicate \textbf{do}
2: \quad \( P \leftarrow \{ \text{exp}_i[X, Y] \mid X \text{ is distinguished} \}; \)
3: \quad \textbf{if} \( P \) is empty \textbf{then}
4: \quad \quad \text{arbitrarily choose untreated expression predicate } \text{exp}[Z, W];
5: \quad \quad \text{create } B_Z \text{ with all the nodes in the document;}
6: \quad \quad \text{populate}(B_Z, \text{exp}[Z, W]); \ //\text{orphan predicate}
7: \quad \quad \text{mark } \text{exp}[Z, W] \text{ as treated;}
8: \quad \textbf{else}
9: \quad \quad \textbf{for each} distinguished variable \( X \) \textbf{do}
10: \quad \quad \quad \text{create } B_X ;
11: \quad \quad \quad \textbf{for each} \( \text{exp}_i[X, Y] \in P \) \textbf{do}
12: \quad \quad \quad \quad \text{populate}(B_X, \text{exp}_i[X, Y]);
13: \quad \quad \quad \quad \text{mark } \text{exp}_i[X, Y] \text{ as treated;}
14: \quad \quad \textbf{end for}
15: \quad \textbf{end for}
16: \textbf{end if}
17: \textbf{end while}
18: \textbf{output} \( \pi_{\overline{V}}((\bigwedge_{1 \leq i \leq n}(\sigma_{\text{const}}Q_i)) \bigodot (\bigwedge_{1 \leq i \leq m}(\text{Exp}_i))); \ //\sigma_{\text{const}} \text{ selects rows with } U_i \text{ constants}

\textbf{procedure} populate (unary relation \( B_X \), expression predicate \( \text{exp} \))
1: \textbf{for each} \( x \) in \( B_X \) \textbf{do}
2: \quad \( R_x \leftarrow \{ r \mid r \text{ is in the result set of } \text{exp} \text{ when evaluated at } x \} \}; \)
3: \quad \( \text{Exp} \leftarrow \text{Exp} \cup \{(x, r) \mid r \in R_x \}; \ //\text{the tuples } (x, r) \text{ are added to } \text{Exp}
4: \textbf{end for}

Algorithm 1: Column-Based On-Demand Evaluation
Input:
An XPath\(^L\) rule:
\[ h(\overline{V}) \leftarrow q_1(U_1), q_2(U_2), \ldots, q_n(U_n), \]
\[ \exp_1[W_{11}, W_{12}], \exp_2[W_{21}, W_{22}], \ldots, \exp_m[W_{m1}, W_{m2}]. \]

Output:
A \(|V|\)-ary relation \(H\) with tuples that represent the result of the query.

Method:
1: while there exist untreated expression predicates do
2: for each distinguished variable \(W\) s.t. \(B_W\) does not exist do
3: create \(B_W\) by projecting treated relations which have \(W\) in their schema on \(W\) and intersecting;
4: end for
5: \(P \leftarrow \{\exp_i[X,Y] \mid X\ \text{is distinguished}\}\);
6: if \(P\) is empty then
7: arbitrarily choose an untreated expression predicate \(\exp[Z,W]\); //an orphan
8: create \(B_Z\) with all the nodes in the XML document;
9: populate\((B_Z, \exp[Z,W])\);
10: mark \(\exp[Z,W]\) as treated;
11: continue;
12: end if
13: choose \(\exp[X,Y]\) from \(P\) s.t. \(|B_X|\) is minimal;
14: populate\((B_X, \exp[X,Y])\) //see procedure populate below;
15: \(B_X \leftarrow B_X \cap \pi_X(\Exp)\);
16: if \(B_Y\) exists then
17: \(B_Y \leftarrow B_Y \cap \pi_Y(\Exp)\);
18: end if
19: mark \(\exp[X,Y]\) as treated;
20: end while
21: output \(\pi_{\overline{V}}((\bigotimes_{1 \leq i \leq n} (\sigma_{\text{const} Q_i})) \bigotimes ((\bigotimes_{1 \leq i \leq m} (\exp_i))))\);

Algorithm 2: Improved Column-Based On-Demand Evaluation
**Improving Column-Based On-Demand.** There are a few possible improvements to Column-Based On-Demand for conjunctive queries:

- In Algorithm 1, expression relations whose first argument is the same variable, are populated with the same binding column. Consider an expression predicate $exp[X, Y]$. Observe that in the corresponding expression relation $Exp[X, Y]$, $\pi_X(Exp) \subseteq B_X$ since evaluating $exp$ on elements of $B_X$ may have an empty result set. Therefore, after populating $Exp$, we can update $B_X$, i.e., $B_X \leftarrow B_X \cap \pi_X(Exp)$, which will save calls to the XPath processor in the treatment of the next expression predicate whose first argument is the variable $X$ (if one exists).

- In Algorithm 1, we arbitrarily choose one distinguished variable and populate an expression relation with its binding column. It is likely that starting with the smallest cardinality binding column would save work, since an expected small number of nodes would be bound to variables that are second arguments of an expression predicate. The bindings to these variables may be used to create binding columns of future distinguished variables. Small sets of bindings imply potentially small cardinality binding columns.

- In addition to the previous item, we would not necessarily want to populate all the expression predicates with the same first variable consecutively. After an expression relation has been populated, we can update the binding columns of the two variables in the corresponding predicate (i.e., for $exp[X, Y]$, $B_X \leftarrow B_X \cap \pi_X(Exp)$ and $B_Y \leftarrow B_Y \cap \pi_Y(Exp)$ if $B_Y$ exists). Then, based on the new cardinalities of the binding columns, we may choose the smallest one, say $B_W$, and treat an expression predicate whose first argument is $W$.

Pseudo-code with a variant of Column-Based On-Demand that includes these improvements appears in Algorithm 2.

### 2.3.3.2 Join-Based On-Demand

Even though Column-Based On-Demand typically avoids populating expression relations with many unnecessary tuples, it may still populate expression relations with a significant number of unnecessary tuples. Join-Based On-Demand evaluates $XPath^L$ rules after populating expression relations with only (often, proper) subsets of the tuples generated by Column-Based On-Demand.
On the other hand, it performs join operations before populating relations.

**Algorithm Sketch.** The notation used in Section 2.3.3.1 is also used here. Join-Based On-Demand starts by joining all the treated relations, which are at this point the relations $Q_i$, into a relation $J$. $J$ is projected on the attributes of the query’s head and of untreated expression predicates (these variables have a “future use”), and the result is assigned back to $J$. Then, all the expression relations whose first argument is in $J$’s schema are partitioned into two sets, $T_1$ and $T_2$. $T_1$ consists of expression predicates whose first argument is a variable which belongs to $J$’s schema and whose second argument does not. We call these predicates type 1 predicates w.r.t. $J$. $T_2$ consists of expression predicates whose first and second arguments belong to $J$’s schema. We call these predicates type 2 predicates w.r.t. $J$. First, the elements in $T_2$ are considered. Each one of them is populated. Instead of using a binding column, Join-Based On-Demand uses the projection of $J$ on the first attribute of the relation to be populated. The populated relation is joined (in fact, semi-joined) with $J$, and the result replaces $J$. $J$ is again projected on the attributes of the query’s head and of untreated expression predicates, and the result is again assigned to $J$. We start with type 2 predicates because they are likely to eliminate from $J$ more tuples than type 1 predicates. The reason is that given a binding $x_0$ from $J$ for the first argument of a type 1 predicate, its expression only needs to be satisfied at $x_0$ in order not to eliminate tuples with $x_0$ from $J$. However, given a type 2 predicate, say $exp[Z,W]$, and a binding $z_0$ from $J$, in order not to eliminate a tuple from $J$, $exp$ needs to be (1) satisfied at $z_0$ and (2) the result set has to include a value which appears with $z_0$ in the same tuple in $J$. This is done in order to reduce the size of $J$ as much as possible and as early in the evaluation as possible, because $J$’s projections are used to populate expression relations. A small cardinality $J$ implies less calls of the XPath processor and less tuples materialized in joins.

When the set $T_2$ is exhausted, we take\(^2\) one predicate from $T_1$, populate it, join it with $J$, project on the attributes of the head as well as those of untreated expression predicates and assign the result back to $J$. Observe that after this join, $T_1$ and $T_2$ have to be recomputed. We again give priority to predicates in $T_2$. If $T_1$ and $T_2$ are both empty, and there are still untreated predicates, we arbitrarily choose an untreated predicate (an orphan) and populate it by evaluating its expression.

\(^2\)The choice may be based on either selection rules or a cost model if either exists, left to future work.
**Input:**
An XPath rule:

\[ h(\bar{V}) \leftarrow q_1(\bar{U}_1), q_2(\bar{U}_2), \ldots, q_n(\bar{U}_n), \exp_1[W_{11}, W_{12}], \exp_2[W_{21}, W_{22}], \ldots, \exp_m[W_{m1}, W_{m2}] \]

**Output:**
A \(|\bar{V}|\)-ary relation \(H\) with tuples that represent the result of the query.

**Method:**
1. \(J \leftarrow\) join of all the relations corresponding to predicates \(q_i\);
2. \(J \leftarrow\) project\((J)\); //see procedure project below
3. while there exist untreated predicates do
   4. compute \(\{\exp_i[X, Y] \mid X \text{ is in } J\'s \text{ schema}\}\) and partition to \(T_1\) (if type 1) and \(T_2\) (if 2);
   5. while \(T_2\) is not empty do
      6. arbitrarily remove an element \(\exp[X, Y]\) from \(T_2\);
      7. populate\(\pi_X(J),\exp[X, Y]\); mark \(\exp[X, Y]\) as treated; \(J \leftarrow\) project\((J \Join\ Exp)\);
   8. end while
9. if \(T_1\) is not empty then
   10. arbitrarily remove an element \(\exp[X, Y]\) from \(T_1\);
   11. populate\(\pi_X(J),\exp[X, Y]\); mark \(\exp[X, Y]\) as treated; \(J \leftarrow\) project\((J \Join\ Exp)\);
12. else
13. arbitrarily choose an expression predicate \(\exp[X, Y]\);
14. create \(B_X\) with all the nodes in the XML document;
15. populate\(B_X,\exp[X, Y]\); mark \(\exp[X, Y]\) as treated; \(J \leftarrow\) project\((J \Join\ Exp)\);
16. end if
17. end while
18. output \(\pi_{\bar{V}}(J)\)

**procedure** project (relation \(J\))

1. \(A \leftarrow\) \{variables in \(\bar{V}\)\} \(\cup\) \{variables in untreated expression predicates\};
2. return \(\pi_{a \in A}(J)\);

**Algorithm 3: Join-Based On Demand Evaluation**
at every node in the tree. The resulting relation is joined with \( J \), \( T_1 \) and \( T_2 \) are recomputed, and the algorithm continues until all predicates are treated.

Pseudo-code for Join-Based On-Demand appears in Algorithm 3.

2.3.3.3 Quick On-Demand Evaluation

Quick On-Demand consists of improvements to Join-Based On-Demand, which are aimed at reducing the number of tuples materialized during evaluation.

The relational untreated predicates graph. Given an XPath\(^{\uparrow}\) rule, we define the relational untreated predicate graph (RUP) as follows. Every untreated relational predicate is a node. An undirected edge exists between two nodes if they share at least one variable. We associate each connected component in the graph with the set of expression predicates whose first argument is a variable which appears in one of the relational predicates in the component.

Algorithm Sketch. The notation used in Sections 2.3.3.1 and 2.3.3.2 is also used here. Given a rule \( r \), all predicates are initially marked as untreated. Quick starts by building the relational untreated predicates graph. Then, using an abstract algorithm \( A \), which is an input to Quick, a set of nodes \( c_0 \), whose elements are the nodes in one connected component, is extracted from the graph. We require that \( A \) builds the set only from a component that is associated with a non-empty set of expression relations. Then, a rule \( r_0 \) is built as follows. \( r_0 \)’s body consists of the relational predicates in \( c_0 \) and the associated expression predicates. \( r_0 \)’s head consists of all the variables which appear in the body. Relational predicates in \( r_0 \) are marked as treated, and \( r_0 \) is evaluated using Join-Based On-Demand. We call the result \( J_0 \) (we henceforth call the result of the \( i \)-th call to Join-Based On-Demand - \( J_i \), starting from \( i = 0 \). Also, attributes names in \( J_i \) are the names of the corresponding variable in \( r_i \)’s head). \( J_0 \) is projected on all the attributes in untreated predicates. Note that at this point, untreated predicates are actually all the predicates which are not a part of \( r_0 \)’s body, as Join-Based On-Demand returns after all the predicates are treated. We call the result of the latter projection \( K_0 \). \( k_0 \), a predicate with parameters corresponding to \( K_0 \) is added to \( r \) (marked as untreated). The relational untreated predicates graph is built again, a set of nodes \( c_1 \) (corresponding to another connected component) is extracted, and a rule \( r_1 \) is built and sent to
**Input:**
- An XPath$^L$ rule:
  \[ h(\bar{V}) \leftarrow q_1(\bar{U}_1), q_2(\bar{U}_2), ..., q_n(\bar{U}_n), \text{exp}_1[W_{11}, W_{12}], \text{exp}_2[W_{21}, W_{22}], ..., \text{exp}_m[W_{m1}, W_{m2}] \]
- An Algorithm $A$ which extracts from an input graph nodes belonging to a connected component that has a non-empty set of expression relations associated with it.

**Output:**
A $|\bar{V}|$-ary relation $H$ with tuples that represent the result of the query.

**Method:**

1: $i \leftarrow 0$

2: while there exist untreated predicates do

3: build the relational untreated predicates graph $G$;

4: if there are untreated expression predicates but none of the components in $G$ has expression predicates associated with it then

5: arbitrarily assign an expression predicate to one of the components; //orphan

6: $c \leftarrow A(G);$ //choose a component

7: end if

8: if all the untreated predicates are relational then

9: $c \leftarrow$ all untreated relations; //note that they are all relational

10: end if

11: build rule $r_i$ whose body consists of the predicates in $c$ and the expression predicates associated with $c$ (if any). $r_i$’s head consists of all the variables in the body;

12: $J_i \leftarrow$ Join-Based On-Demand($r_i$); //JBOD marks predicates as treated

13: create new relation $K_i$ from $J_i$ and add the corresponding predicate $k_i$ to the original rule, marked as untreated; $i \leftarrow i + 1$;

14: end while

15: $l \leftarrow i - 1$

16: while $l > 0$ do

17: $J_{l-1} \leftarrow$ project2($J_l \bowtie J_{l-1}$); $l \leftarrow l - 1$;

18: end while

19: output $\pi_{\bar{V}}(J_0)$

**Algorithm 4: Quick On Demand Evaluation**
Join-Based On-Demand. The result, \( J_1 \), is projected to produce \( K_1 \), and the corresponding predicate \( k_1 \) is added to \( r \). The graph is built again, and so on until all the predicates are marked as treated.

**Remarks:**

1. If at any point none of the connected components has expression predicates associated with it but there are untreated expression predicates (orphans), one connected component and one orphan predicate are artificially associated and the algorithm continues.

2. If at any point none of the connected components has expression predicates associated with it and all the expression predicates are treated, a rule \( r_{\text{relational}} \) is built. \( r_{\text{relational}} \)'s body consists of all the untreated predicates. \( r_{\text{relational}} \)'s head consists of all the variables in the body. \( r_{\text{relational}} \) is evaluated using Join-Based On-Demand. After evaluation, no untreated predicates exist.

Let us define the operation \( \text{project}_2 \) on a relation \( J_i \) where \( i > 0 \). \( \text{project}_2(J_i) \) is the projection of \( J_i \) on the union of the following attribute sets:

1. Variables in the original rule’s head.

2. Attribute names in \( J_{i-1} \).

For each of the relations \( J_i \), starting with the relation where \( i \) is maximal, we perform \( \text{project}_2(J_i) \), and join the result with \( J_{i-1} \). We assign the result to \( J_{i-1} \) and continue. When reaching \( J_0 \), instead of performing \( \text{project}_2 \) (note that \( \text{project}_2 \) is undefined for \( J_0 \)), we project \( J_0 \) on the head variables. The result is added to the relation corresponding to the head’s predicate. This project and join procedure which is performed on the relations \( J_i \) is similar to Yannakakis’ algorithm for taking the projection of a join of relations with an acyclic hypergraph corresponding to their schemes [92]. In our case, we do not use the GYO-reduction to construct a tree from the \( J_i \) relations, but consider the \( J_i \) in the reverse order to the one of creating them. Note that the schema of \( J_i \)'s is such that once an attribute is in \( J_i \) and not in \( J_{i+1} \), the attribute will not appear in any \( J_k \) s.t. \( k > i + 1 \). Therefore, projecting each \( J_i \) on the variables in the head and on the variables in \( J_{i-1} \) is sufficient to ensure that the joins produce the same tuples that would have been produced had all the \( J_i \) been joined without any projection (a costly procedure).

Pseudo-code for Quick On-Demand appears in Algorithm 4.
**Orphan Expression Predicates.** Column-Based and Join-Based On-Demand treat orphan predicates by arbitrarily choosing one such predicate, if one exists, and proceed with the algorithm. However, different choices as to which orphan to handle first can lead to different behaviors as demonstrated next.

**Example.** Consider the following query, and an arbitrary database instance:

\[ \text{res}(Z) \leftarrow a/b[W,Z], b/c[Z,X] \]

Both predicates in the body are orphans. Suppose that \( a/b[W,Z] \) is treated first, and \( a/b \) is evaluated at all the nodes in the document as context nodes. In Column-Based On-Demand, a binding column \( B_Z \) is created because of the untreated predicate \( b/c[Z,X] \), and the corresponding relation is populated using \( B_Z \). However, if \( b/c[Z,X] \) is treated first, then there is no binding column for \( W \), and the relation corresponding to \( a/b[W,Z] \) is populated by evaluating \( a/b \) at all the nodes in the XML document. A similar behavior is observed in Join-Based On-demand. If \( a/b[W,Z] \) is treated first, then \( Z \) is in \( J \)'s schema and populating \( b/c[Z,X] \) does not necessitate evaluating \( a/b \) at all the nodes in the document (unlike the other case).

We use the following method in order to avoid, as much as possible, evaluating expressions at every node in the tree. An *orphan graph* is constructed. Every orphan predicate is a node. There is a directed edge from orphan \( o_1 \) to orphan \( o_2 \) if \( o_1 \)'s second argument is the same as \( o_2 \)'s first argument. For example, the orphan graph for the example above is:

\[ a/b[W,Z] \rightarrow b/c[Z,X]. \]

Then, the orphan graph is topologically sorted. We populate a relation corresponding to a source of the sorted graph which has the largest number of descendants, i.e., a heuristics, and proceed with the main algorithm. Note that after populating the source, the algorithm need not treat the descendants as orphans any longer.

**Example.** In the example above, \( a/b[W,Z] \) is the only source. After it is populated, \( b/c[Z,X] \) is no longer an orphan, since \( Z \) has a binding column (in Column-Based) or appears in \( J \)'s schema (in Join-Based).
This approach can also be generalized to Quick On-Demand. Instead of associating an arbitrary orphan to a connected component (line 5 in Algorithm 4), we associate a selected source of the topologically sorted graph.

2.3.3.4 Conjunctive Queries with Negated predicates

Up to this point we treated queries without negation. In the presence of negation, predicates can be partitioned to *expression* and *relational* as well as to *positive* and *negative* (a total of four possibilities). We consider queries in which there are no negated predicates having variables which do not appear in positive predicates (these can be trivially satisfied). We devise a version for each of the algorithms which populates expression relations in the presence of negation.

**Column-Based On-Demand with Negation.** We describe the addition of negation to the improved version of Column-Based On-Demand (i.e., Algorithm 2). First, we generalize the definition of a distinguished variable for queries with negation. Variable $X$ is distinguished at a certain point in time if $X$ is a first variable in at least one untreated (positive or negative) expression predicate and $X$ appears as an argument in a *positive* treated predicate. This change in definition may cause the addition of negated predicates to $P$, the set of predicates (used in Algorithm 2). The creation of a binding column for a variable $X$ is done by projecting only *positive* treated relations on $X$, as opposed to Algorithm 2, in which we defined the creation of the binding column as done using projections from all the treated relations on $X$. Line 13 in Algorithm 2 is changed so that the choice of a predicate from $P$ can be of a negated or a non-negated predicate.

In Algorithm 2, line 21, the relations corresponding to positive predicates are joined. Let us denote by $I$ the result of this join. In the presence of negation, after joining the predicates corresponding to positive relations and producing $I$, every relation $R$ corresponding to a negated predicate is considered. For each such relation we update $I$: $I \leftarrow I \bowtie (\pi_{\text{attr}(R)}(I) \setminus R)$, where $\text{attr}(R)$ is the set of $R$’s attributes. Then, the result is output as a projection.

The rest of the algorithm remains unchanged. The execution result is that an expression relation $\text{Exp}$ is populated if predicates $\neg \text{exp}[X,Y]$ or $\text{exp}[X,Y]$ appear in the body, based on the values in the binding column for $X$, as usual. We argue that this population leads to a correct evaluation.
of the rule, even though only a part of $Exp$ is materialized. This is true since $B_X$ contains all the values of $X$ based on which tuples may be added to the relation corresponding to the rule’s head ($B_X$ is the intersection of projections on $X$ of positive treated relations). Therefore, given a predicate $\neg exp[X, Y]$, $Exp$ need not be checked for the existence or absence of 2-tuples whose first element is not in $B_X$, because such tuples are necessarily not a part of a satisfying assignment to the variables in the rule’s body, and will not produce tuples to be added to the relation corresponding to the rule’s head.

**Join-Based On-Demand with Negation.** We define a *bound predicate* to be a predicate whose attributes are all in $J$’s schema. We include in $T_2$ bound, negated expression predicates, in addition to $T_2$’s contents as defined for the positive case. In the absence of negation, after populating a positive expression relation $exp[X, Y]$, $J$ is updated by joining $J$ with $Exp$ and applying the procedure $project$ on the result. If the populated relation corresponds to a negated predicate ($\neg exp[X, Y]$), then the update operation is the following: $J \leftarrow \text{project}(J \bowtie (\pi_{XY}(J) \setminus Exp))$. This update operation is also performed for negated relational predicates which become bound in the course of evaluation.

**Quick On-Demand with Negation.** Evaluation of negation in Quick On-Demand is based on Join-Based On-Demand. During evaluation, Quick ignores negated predicates that are not bound. When a negated predicate becomes bound, Quick considers it to be a part of a connected components in the graph and as a candidate to be associated to such components (in case it is an expression predicate). Bound predicates are therefore sent by Quick to Join-Based On-Demand, to be evaluated as discussed above.

### 2.3.4 On-Demand for Recursive Queries

We turn to the *Column-Based On-Demand* (CBOD) algorithm for multi-rule, and in particular recursive queries, which follows the On-Demand approach for evaluating $XPath^L$ queries. At this point, we restrict attention to queries without negated predicates. Later on, we consider negation.

**Algorithm Sketch.** Pseudo-code for CBOD appears as Algorithm 5 (with related procedures in
**Input:** An XPath\(^L\) program.

**Output:** Populated IDB relations.

**Method:**

1. mark relational predicates as *permanently treated* and empty expression predicates as *temporarily treated*;
2. while fixpoint has not been reached do
   3. for each rule \( r \) in the program do
      4. mark temporarily treated predicates as untreated;
      5. \( \text{extendExpRel}(r); \)
      6. for each relation \( P \) corresponding to an IDB or XPath predicate \( p \) in \( r \) do
         7. let \( \Delta P \) be the tuples added to \( P \) since the start of the previous evaluation of \( r \);
         8. evaluate \( r \) with \( \Delta P \) replacing \( P \);
         9. Add resulting tuples to the relation corresponding to \( r \)’s head predicate;
      10. end for
   11. end for
12. end while

**Algorithm 5: CBOD with Semi-Naive**

The algorithm extends the well known semi-naive evaluation method of solving Datalog queries [92] so as to work efficiently with XPath predicates. A rule \( r \) in a possibly recursive program may be evaluated more than once. Before each evaluation of \( r \), CBOD incrementally populates the expression relations (line 5 in the main algorithm pseudo-code). We mark all the relational predicates \( q_i \) as *permanently treated* (as we will shortly see, predicates may also be marked as *temporarily treated*). Evaluation is started with empty expression relations. CBOD incrementally populates the relations by considering supersets of possible bindings to their first argument. We call a variable \( X \) *distinguished* at a certain point in time if \( X \) is a first variable in at least one untreated expression predicate and \( X \) appears as an argument in a (permanently or temporarily) treated predicate.

The procedure \( \text{extendExpRel} \) is executed before each rule evaluation. For each distinguished variable \( W \), \( W \)'s binding column \( B_W \) is built as follows. We consider the treated relations whose
1: **while** there exists an untreated predicate **do**
2: \[ G \leftarrow \{ \text{exp}_i[X,Y] \mid X \text{ is distinguished} \} ; \]
3: **if** \( G \) is empty **then**
4: arbitrarily choose untreated expression predicate \( \text{exp}[Z,W] \);
5: create \( B_Z \) with all the nodes in the document;
6: populate(\( B_Z, \text{exp}[Z,W] \)); //orphan predicate
7: mark \( \text{exp}[Z,W] \) as *permanently treated*;
8: **else**
9: **for each** distinguished variable \( X \) **do**
10: \( B_X \leftarrow \text{createBindingColumn}(X) ; \)
11: **for each** \( \text{exp}_i[X,Y] \in G \) **do**
12: populate(\( B_X, \text{exp}_i[X,Y] \));
13: mark \( \text{exp}_i[X,Y] \) as *temporarily treated*;
14: **end for**
15: **end for**
16: **end if**
17: **end while**

**Algorithm 6:** extendExpRel(rule \( r \))

---

1: **for each** \( x \) in \( B_X \backslash \pi_X(\text{Exp}) \) **do**
2: \( R_x \leftarrow \{ n \mid \text{exp} \text{ evaluated at } x \text{ yields } n \} ; \)
3: \( \text{Exp} \leftarrow \text{Exp} \cup \{ (x,n) \mid n \in R_x \} ; \)
4: **end for**
5: \( B_X^{\text{dead}} \leftarrow B_X^{\text{dead}} \cup (B_X \backslash \pi_X(\text{Exp})) ; \)

**Algorithm 7:** populate(column \( B_X \), exp. pred. \( \text{exp} \))
1: \( D \leftarrow \{ d \mid d \) is a treated relation with \( X \) in its schema\};
2: \( B_X \leftarrow \emptyset \);
3: for each \( d \) in \( D \) do
4: \( \Delta d \) contain tuples added to \( d \) as of the start of the last iteration of the while loop in Algorithm 5;
5: \( \Delta d \leftarrow \pi_X(\Delta d) \setminus B_{X}^{\text{dead}} \);
6: for each \( c \) in \( D \setminus \{ d \} \) do
7: \( \Delta d \leftarrow \pi_X(\Delta d) \cap \pi_X(c) \);
8: end for
9: \( B_X \leftarrow B_X \cup \Delta d \);
10: end for
11: return \( B_X \);

Algorithm 8: createBindingColumn(variable \( X \))

schema includes \( W \) (distinguished relations). We select, from each such relation at a time, the "new" tuples added to it since the last evaluation of the rule, project them on \( W \) and add to \( B_W \) the intersection of these projections, intersected with the projections of rest of the distinguished relations (in whole) on \( W \). Every expression relation \( Exp_i \) whose first argument is \( W \) is now further populated by evaluating its expression on selected nodes from \( B_W \) as follows. Not all the nodes in \( B_W \) are sent to the XPath processor. Note that \( B_W \) may contain nodes that need not be sent to the XPath processor. These nodes are of two types: (1) nodes on which \( exp_i \) was already evaluated, appearing this time because they are a part of newly produced tuples, whether the result set was empty or not, and (2) nodes on which some other expression, \( exp_j \), corresponding to an expression predicate whose first argument is \( W \), was evaluated, and the result set was empty. Such nodes cannot be a satisfying binding to \( W \) and need not be sent to the XPath processor. Type (1) nodes with a non-empty result set for \( exp_i \) are recorded in \( Exp_i \). Type (1) nodes with an empty result set and type (2) nodes are recorded in a dedicated column, \( B_W^{\text{dead}} \). We do not execute the XPath processor on these nodes. Then, the XPath predicate is marked as temporarily treated, the set of distinguished variables is recomputed and the algorithm continues. If there are no relevant variables but there is an untreated XPath predicate \( p \) (an orphan predicate), the relation corresponding to \( p \) is populated by evaluating \( p \)'s expression on all the nodes in the XML document (this can
happen only once). The predicate is marked as permanently treated, the set of relevant variables is recomputed and the algorithm continues. Before the next time that the procedure is performed, temporarily treated predicates are marked as untreated.

Binding columns are reminiscent of the supplementary relations of the Query-Sub-Query Datalog optimization technique, since they limit the number of nodes sent to the XPath processor. However, unlike Query-Sub-Query [1], they are formed by intersecting over all relevant projections rather than in a left-to-right order, and also do not include previously failed activations on expression relations.

Negation. We generalize CBOD to evaluate queries with negated predicates. We assume that queries are stratified [92], and consider the strata in a bottom-up fashion. In each stratum, we generalize the definition for a distinguished variable for queries with negation. X is distinguished at a certain point in time if X is a first variable in at least one untreated expression relation and appears as an argument in a positive treated predicate. This may cause the addition of negated predicates to G (the set of expression predicates whose first argument is distinguished). Line 11 in Algorithm 6 is therefore changed to iterate also on negated predicates in G. Second, the creation of a binding column is done by including only positive relations in D (in Algorithm 8). The rest of CBOD remains unchanged. The result is that an expression relation Exp is populated if predicates \( \neg \text{exp}[X,Y] \) or \( \text{exp}[X,Y] \) appear in the body. Third, we adjust Algorithm 5 to handle negated predicates by changing Line 6 to consider only positive predicates \( p \) in \( r \). We argue that such expression relation population leads to a correct evaluation of the rule, since the elements in \( B_X \) (Line 11, Algorithm 8) contain all the values of \( X \) based on which new tuples may be added to the relation corresponding to the rule’s head. Therefore, given a predicate \( \neg \text{exp}[X,Y] \), \( \text{Exp} \) need not be checked for the absence of two-tuples whose first element is not in \( B_X \), because such tuples in \( \text{Exp} \) necessarily do not support the creation of any new tuple. Since after the call to populate, no new tuples whose first element is in \( B_X \) are added to \( \text{Exp} \) in this call to \( \text{ExtendExpRel} \), the evaluation based on the materialized part of \( \text{Exp} \) is correct.

On-Demand for \( \text{Val} \). The On-Demand strategy for populating \( \text{Val} \) uses the hash codes discussed in Section 2.3.2 in order to minimize checking for isomorphism. We extend the definition
given to a distinguished variable in the context of expression relations. Given an untreated pred-
icate \( val(X, Y) \), \( X \) (respectively, \( Y \)) is distinguished at a certain point in time if there exists a
treated relation with \( X \) (respectively, \( Y \)) in its schema. Algorithm 6 is extended. The set \( G \) is
created with \( val \) predicates with at least one distinguished variable. Binding columns for distin-
guished variables are created. Assuming a predicate \( val(X, Y) \), \( Val \) is populated is one of two
"directions", either according to \( B_X \) or to \( B_Y \). Consider an element \( x \) in \( B_X \). If \( x \)'s hash code
was not yet encountered, \( x \) is assigned a new value id and is inserted with the value id into \( Val \).
Otherwise, isomorphism is checked with the other elements that have the same hash code. If an
isomorphism to, say \( x' \), is found, then \( x \) is assigned the value id already associated with \( x' \). In the
other direction, i.e., if the binding column \( B_Y \) for the second argument is given, isomorphism is
checked for all the nodes which have \( y \)'s hash code. \( Val \) is populated accordingly. This way, only
values "demanded" by a program are recorded in \( Val \).

### 2.4 System Prototype, Algorithm Implementation and Experi-
mentation

#### 2.4.1 System Prototype

We designed and implemented a system prototype to experiment with the Static and On-Demand
evaluation approaches and algorithms. Figure 2.4 depicts the run-time system architecture. Next,
we describe the functionality of its components.

**The Parser.** We represent a query using an XML document which allows the easy creation, by
using an XPath processor, of the data structures which represent the executed program.

**Built-In Relations Loader.** This component is responsible for creating relation \( Val \) (for predic-
cate \( val \)) and the relations which correspond to XPath predicates. It populates these relations with
tuples, using the **XPath Processor**. Populating is done using either the **Static** or **On-Demand** ap-
proach.
Conceptually, there are infinitely many built-in relations (one for every possible expression, called *expression relations*). However, with respect to a particular XML document and one query, there is a finite number of such relations, and each relation has a finite number of tuples. If $m$ is the number of nodes in the XML document, $Val$ has $m$ tuples in which the first element is an id of a node of the original document, and an expression relation has at most $m^2$ such tuples. In addition, $Val$ and expression relations have tuples in which the first element is one of the $O(m^2)$ objects that are descendants of value objects. As discussed above, the Static evaluation approach creates whole relations, while the On-Demand approach populates these relations based on known binding to variables at run time.

**SQL Interface.** This component executes relations-related operations. The built-in relations loader uses this interface to load tuples into the built-in relations, and the evaluation manager (discussed next), uses this interface to query and update relations.

**Evaluation Manager.** This component handles the evaluation process. First, it creates the IDB relations (initially empty). Then, it evaluates the (possibly stratified) query by considering the strata in a bottom up fashion, from lowest to highest, according to one of the algorithms. The evaluation is done using one of the algorithms described above. Note that CQs do not have strata.
**SQL Generator.** The SQL generator creates the queries which retrieve the satisfying bindings for the variables of the rule and updates the relevant relations using these bindings. The output of a query program, as well as intermediate results, is simply contained in table(s) in the relational DBMS.

### 2.4.2 Implementation and Experimentation

For conjunctive queries, we implemented the CBOD, Join and Quick algorithms (with an $A_1$ as described in Section 2.3.3 and an $A_2$ that chooses the component to which the leftmost untreated relational predicate belongs). For recursive queries, we implemented the Static algorithm and the CBOD for recursive queries.

The implementation is built using off-the-shelf components: (1) A Java-written application; (2) The open-source relational database system Derby (version 10.3) [29] and (3) the Jaxen XPath evaluator (version 1.1.1) [59]. We also translated the queries with which we experimented to XQuery. In our experimentation, we compare the running time of CBOD, Quick and Saxon’s XQuery engine (SX) [86]. For CBOD and Quick, we also check the number of tuples materialized in expression relations. The experiments were carried out on a Lenovo ThinkCentre with a Pentium-M, 1.6GHZ CPU and 2GB of RAM, running the Windows XP-Professional operating system. The application implementing the algorithms was invoked using an Eclipse JVM with a maximal Java heap size tuned to 1400MB.

#### 2.4.2.1 Data and Queries

**Databases.** We conducted our experiments using hybrid XML-Relational databases that we created from XMark generated XML documents [103]. Eight relations were created from each XMark document. Their contents are ids of XML elements and text (i.e., strings that are not ids). The relation names and contents description appear in Figure 2.5. In order to experiment with XQuery, we also transformed these relations to XML documents. An XML document representing a relation, say $r$, is structured as follows. The root element has tag `relation` and all its direct children have tag `tuple`. If $r$ is a $k$-ary relation, each `tuple` element has $k$ children with tag `col`. For $1 \leq i \leq k$, the $i$’th `col` element has attribute `val` whose value is the content of the $i$’th column in the tuple. For
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rel-a(·)</td>
<td>About 33% of the open_auction elements, randomly chosen, and text.</td>
</tr>
<tr>
<td>Rel-b(·,·)</td>
<td>All open_auction elements (1st column), text, and a random closed_auction element (2nd column).</td>
</tr>
<tr>
<td>Rel-c(·,·)</td>
<td>Text, all closed_auction elements, and a random person element.</td>
</tr>
<tr>
<td>Rel-d(·,·)</td>
<td>All address elements, each with the person element which immediately follows the address element’s parent (a person element), and text.</td>
</tr>
<tr>
<td>Rel-e(·,·)</td>
<td>All profile elements, text, and all the profile’s interest child elements.</td>
</tr>
<tr>
<td>Rel-f(·,·)</td>
<td>Text, all interest elements, each with a random person element.</td>
</tr>
<tr>
<td>Rel-g(·)</td>
<td>Text and about 33% of the open_auction elements, randomly chosen.</td>
</tr>
<tr>
<td>Rel-h(·,·)</td>
<td>All open_auction elements, each with a random person element, and text.</td>
</tr>
</tbody>
</table>

Figure 2.5: Benchmark Relations and their contents.

example, the XML form of relation Rel-a, in which every tuple is of the form (open_auction id, text), is illustrated next.

```xml
<relation>
  <tuple>
    <col value="2345"/>
    <col value="abcdefg"/>
  </tuple>
  <tuple>
    <col value="98765"/>
    <col value="thkhaya"/>
  </tuple>
  ....
  ....
</relation>
```


**Queries.** The queries with which we experimented are of three types, Chain, Zigzag and Mixed. They are shown, organized by type, in Figure 2.14. Some queries are a cyclic variation of another query (a query q is cyclic if the undirected graph, whose nodes are q’s predicates and in which an edge exists between every two predicates that share a variable, is cyclic). We only refer to ids and
Figure 2.6: Results for Q1.

Figure 2.7: Results for Q2.

Figure 2.8: Results for Q3.
Figure 2.9: Results for Q4.

Figure 2.10: Results for Q5.

Figure 2.11: Results for Q6. XQuery did not return in reasonable time.
Figure 2.12: Results for Q7.

<table>
<thead>
<tr>
<th>Query</th>
<th>Over population factor of CBOD, in percentage, for DBs 1 through 5.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>24 33 28 25 25</td>
</tr>
<tr>
<td>Q2</td>
<td>35 50 41 36 36</td>
</tr>
<tr>
<td>Q3</td>
<td>57 93 74 N/A N/A</td>
</tr>
<tr>
<td>Q4,5,6</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>Q7</td>
<td>57 50 59 48 50</td>
</tr>
</tbody>
</table>

Figure 2.13: Over Populating Factor values.

Figure 2.14: Queries used in the experiments (a).

Chain:
(1) h(X,N)←rel-a(X,_,), rel-b(X,_,Y), rel-c(_,Y,Z), descendant::address[Z,M], child::country[M,N].
(2) h(X,N)←rel-a(X,_,), rel-b(X,_,Y), rel-c(_,Y,Z), descendant::address[Z,M], child::country[M,N], following::open_auctions/open_auction[1][N,T].
(3) h(X,N)←rel-a(X,_,), rel-b(X,_,Y), rel-c(_,Y,Z), descendant::address[Z,M], child::country[M,N], following::open_auctions/open_auction[N,X].

Zigzag:
(4) h(X,P)←rel-c(_,X,Y), child::address[Y,W], rel-d(W,Q,_), child::profile(Q,P), rel-e(P,_,I).
(5) h(X,P)←rel-c(_,X,Y), child::address[Y,W], rel-d(W,Q,_), child::profile(Q,P).
(6) h(X,P)←rel-c(_,X,Y), child::address[Y,W], rel-d(W,Q,_), child::profile(Q,P), rel-e(P,_,I), rel-f(_,I,Y).

Mixed:
(7) h(Y,Q)←rel-g(_,X), self::person[name][Y,Q], fol-sib::person[position<=3][Q,W], descendant::profile[W,P], rel-e(P,_,I).
Figure 2.15: Queries used in the experiments (b).

not to text in our queries. Arguments corresponding to text columns are therefore marked with "._". For example, rel-a(X,_).

Translation to XQuery. The queries were translated to equivalent XQuery queries over the original XML and the additional XML documents which represent the relations. For queries 1, 2, 3 and 7, we translate to XQuery. For example, query 1 is translated to:

```xquery
for $x1 in doc(rel-a.xml)/relation/tuple/col[1],
  $x2 in doc(rel-b.xml)/relation/tuple/col[1],
  $y2 in doc(rel-c.xml)/relation/tuple/col[2],
  $z2 in doc(benchmark.xml)//*[attr::id],
  $m2 in doc(benchmark.xml)//*[attr::id]
let $m1 := $z2/descendant::address,
  $n1 := $m2/child::country,
  $y1 := $x2/following-sibling::col[2],
  $z1 := $y2/following-sibling::col[1]
where ($x1/attr::value=$x2/attr::value and
  $y1/attr::value = $y2/attr::value and
  $z1/attr::value = $z2/attr::id and
  $m1/attr::id = $m2/attr::id)
return fn:concat($x1/attr::value","",$n1/attr::id);
```

For queries 4 and 5, the XQuery executions with this naive translation did not terminate within reasonable times. We therefore added information to the XQuery translation (henceforth called translation 2), by restricting each XQuery variable which represents a first variable in an expression predicate to be bound to the element type implied by the relations’ schemas. For example, the schema for Rel-c indicates that the third column is restricted to person elements. Therefore, the XQuery variable corresponding to Z in query Q1 was restricted to be bound to person elements,
as can be seen in the translation for query Q4:

```xml
for $x1 in doc(rel-c.xml)/relation/tuple/col[2],
        $w2 in doc(rel-d.xml)/relation/tuple/col[1],
        $p2 in doc(rel-e.xml)/relation/tuple/col[1],
        $y2 in doc(benchmark.xml)//person,
        $q2 in doc(benchmark.xml)//person
let $y1 := $x1/following-sibling::col[1],
    $p1 := $q2/child::profile,
    $w1 := $y2/child::address,
    $q1 := $w2/following-sibling::col[1],
    $i1 := $p2/following-sibling::col[2]
where ($y1/attr::value = $y2/attr::id and
    $w1/attr::id = $w2/attr::value and
    $q1/attr::value = $q2/attr::id and
    $p1/attr::id = $p2/attr::value)
return fn:concat($x1/attr::value,",",$p1/attr::id);
```

The results of translation 2 are marked `xquery-2` in the figures. For query Q6, both translations did not terminate within a reasonable time, and do not appear in the corresponding graph.

### 2.4.2.2 Results and Discussion

Figures 2.6 through 2.12 depict the running times of CBOD, Quick and SX (in sec.) against the size of the XML document (in MB). Figure 2.13 also shows a table with over populating factor (`o.p.f`) values, explained next. Points which do not appear in the graphs imply that the algorithm(s) did not terminate within a reasonable time. Next, we discuss the results.

**Q1.** Both CBOD and Quick outperform SX. CBOD is slightly faster, even though it populated expression relations with 24-33% more tuples than Quick (we henceforth call this percentage the `o.p.f. of CBOD`). Apparently, this saving did not balance the overheads of Quick.

**Q2.** Here, the `o.p.f.` ranges between 35% and 50%. For DB1-4, CBOD outperforms Quick due to the same reasons as in Q1. In DB5, large joins dominate the execution time. Because of the optimized join treatment, Quick is faster in this case. The time for SX on DB4 is 10 times longer.
than for CBOD. The corresponding point is omitted from the graph so as not to make it too zoomed out. A dotted arrowed line indicates the trend towards this point. This particular experiment was run on a different machine\(^2\).

**Q3.** The o.p.f. ranges between 57\% and 93\%, due to the treatment of the first component of the RUP by Quick after populating the first expression relation. This gives Quick a clear superiority over CBOD. The time for SX on DB3 is 3 times longer than for CBOD. The corresponding point is omitted and the trend towards this point is indicated.

**Q4-5.** For these queries we used translation 2. For both queries, there is no over-populating by CBOD (that is, both Quick and CBOD materialize the same number of tuples). Therefore, CBOD performs slightly better, being a simpler algorithm. In Q4, both algorithms significantly outperform SX, despite the use of translation 2. In Q5, SX is faster.

**Q6.** Due to the same reasons as in Q4-5, CBOD outperforms Quick. SX did not return within a reasonable time (for both types of translation).

**Q7.** The time for CBOD on DB5 is 10 times longer than for Quick, and the time for SX on DB4 is 6 times longer than for CBOD. These points are omitted, and the trend towards them indicated. In this case, expression relations contain large numbers of tuples which imply a lot of time spent on XPath evaluation. Therefore, Quick outperforms CBOD. Both outperform SX.

### 2.4.3 Recursive Queries

For recursive queries, we compared the On-Demand and Static query evaluation using XMark generated XML data [103]. We used XML documents of sizes 1.65MB, 5MB, 8.66MB, 11.7MB, 14.3M, 17.7MB, 23.7MB and 34.5MB. We built recursive XPath\(^L\) queries with XPath and relational predicates. The XPath expressions inspired by the XPathMark queries benchmark [49] (which is not concerned with recursive queries). The experiments were carried out on an IBM machine with a 1.5 GHz Pentium M processor and 1GB of RAM. The recursive queries (listed in Figure 2.15) partition selected nodes to relations even and odd according to the parity of their relative position among their siblings or transitively find all sibling nodes using the closure of the following-sibling[1] relation. The results appear appear in the graphs in Figures 2.16 and 2.17.

For each query, a graph describes the evaluation time against the size of the database in MB. Ob-

\(^2\)An IBM machine with a 1.5 GHz Pentium M processor and 1GB of RAM running Windows XP Professional.
serve that in some cases, static evaluation is faster than On-Demand. However, On-Demand runs significantly faster for the larger databases. The results show the superiority of CBOD, as well as its scalability.

**Hybrid Static and On-Demand Evaluation.** Future work in the area may combine static and On-Demand tuple generation in a hybrid approach, based on factors such as the document’s size and expression selectivity.

### 2.4.4 Experiments with Populating Val

We tested the technique for populating Val, discussed in Section 2.3.2 against XMark generated XML data [103]. In the tests, we considered only the basic problem of assigning values to the nodes in the original tree (and did not consider nodes within values). The graph in Figure 2.18 shows the running time of populating Val with tuples as a function of $n$, the number of nodes in the XML tree. The machine on which this experiment was carried out is an IBM machine with a
Figure 2.18: Results of experiments with populating Val.

1.5 GHz Pentium M processor and 1GB of RAM. The graph demonstrates a linear behavior. This performance cannot be asymptotically improved, since even with the best case scenario, the need to populate Val with $n$ tuples gives the trivial lower bound of $\Omega(n)$ for such a procedure. In Section 2.5.2, we presented a bound of $\Theta(n + L)$ for assigning tags to $n$ nodes such that two nodes are assigned the same tag iff they have the same value, where $L$ is the total length of the tags and texts in the document.

2.5 Theoretical Results

2.5.1 Undecidability of XPath$^L$ Satisfiability

Theorem 2.1 The problem whether or not there exists an XML document (or, XML database) which satisfies an XPath$^L$ query $G$ is undecidable.

Proof: We use a reduction from the undecidable Post Correspondence Problem (PCP) [74]. Assume, for the purpose of deriving a contradiction, that satisfiability of XPath$^L$ queries is decidable. Let $P$ be a PCP instance. Based on $P$, we construct $G$, an instance of the XPath$^L$ satisfiability problem (i.e., a query), and prove two propositions:

- According to Proposition 2.2, if $G$ is satisfiable, then $P$ has a solution.
- According to Proposition 2.3, if $P$ has a solution, then $G$ is satisfiable.
By this, we show that the PCP is decidable, a contradiction. The contradiction implies that satisfiability of XPath$^L$ is undecidable.

In order to prove the following propositions, let us denote the components of $P$ as follows:

$$P = (L, R)$$

where $L = (l_1, l_2, ..., l_n)$ and $R = (r_1, r_2, ..., r_n)$ such that $l_i$ and $r_i$ are each a string over alphabet $\Sigma$ such that $|\Sigma| \geq 2$.

Given $P$, $G$ is built from the following rules ($res()$ is the target predicate):

- An initialization rule:
  $$q(X, X) \leftarrow \text{root}(X).$$

- Let us denote the characters in the strings $l_i$ and $r_i$ as follows. $l_i = l_1^l r_1^l l_2^l r_2^l ..., l_i^l$, $r_i = r_1^r r_2^r ..., r_i^r$. For each $i$, $1 \leq i \leq n$ (i.e., for each pair $l_i, r_i$), a step rule:
  $$q(X, Y) \leftarrow q(W_1, Z_1), \quad \text{child}:: l_1^i [W_1, W_2], \quad \text{child}:: l_2^i [W_2, W_3], ..., \quad \text{child}:: l_i^i [W_i, X], \quad \text{child}:: r_1^i [Z_1, Z_2], \quad \text{child}:: r_2^i [Z_2, Z_3], ..., \quad \text{child}:: r_i^i [Z_i, Y].$$

- A final rule:
  $$res() \leftarrow \text{root}(W), \text{descendant}:: *[W, X], \ q(X, X).$$

The initialization rule inserts a binary tuple in which each element is the id of the root element into the relation $Q(\cdot, \cdot)$.

Let $q(w_0, z_0)$ be a ground subgoal in the body of an instance of step rule $i$. Step rule $i$ descends from $w_0$ to bindings for $X$, along a path of nodes in which the $j$-th node is tagged by $l_j^l$ (i.e., the $j$-th character of the string $l_i$). Similarly, step rule $i$ descends from $z_0$ to bindings for $Y$, along a path of nodes in which the $k$-th node is tagged by $r_k^r$.

The final rule inserts the empty tuple to $Res()$ if there is a tuple in $Q(\cdot, \cdot)$ such that both its elements are the same id, but not the id of the root.
Proposition 2.2 If $G$ is satisfiable, then $P$ has a solution.

**Proof:** If $G$ is satisfiable, then there exists an XML database in which there is a path $p$ from the root to one of its proper descendants, say $n_0$, such that the tuple $(n_0, n_0)$ exists in $Q$.

Since $n_0$ is bound to the first element in the tuple, there exists a sequence $i_1 i_2 \ldots i_k$ such that the result of concatenating the strings $l_{i_1}, l_{i_2}, \ldots, l_{i_k}$ (in this order), is a string $S_l$ such that the character in position $k$ in $S_l$ is also the tag of the $k$-th node in the path $p$.

Since $n_0$ is bound to the second element in the tuple, the same sequence $i_1 i_2 \ldots i_k$ is such that the result of concatenating the strings $r_{i_1}, r_{i_2}, \ldots, r_{i_k}$ (with respect to this order), is a string $S_r$ such that the character in position $k$ in $S_r$ is also the tag of the $k$-th node in the path $p$.

Therefore, $S_l = S_r$, and $i_1 i_2 \ldots i_k$ is a solving sequence for $P$.

Proposition 2.3 If $P$ has a solution, then $G$ is satisfiable.

**Proof:** If $P$ has a solution, then there exists a solving sequence $i_1 i_2 \ldots i_k$ for $P$, which implies that $l_{i_1} l_{i_2} \ldots l_{i_k} = r_{i_1} r_{i_2} \ldots r_{i_k}$. Let us denote $S = l_{i_1} l_{i_2} \ldots l_{i_k}$.

Consider an XML database in which there is one root-to-leaf path (where the root is the one direct child of the document root element), such that the tag of the $k$-th node is the character in the $k$-th position of $S$. Denote the leaf of this path by $n_0$.

We argue that the fact $q(n_0, n_0)$ is derived by the program. The corresponding evaluation of $G$ is as follows. Evaluating the initialization rule, the step rules corresponding to the sequence $i_1 i_2 \ldots i_k$ and the final rule. Since $S = l_{i_1} l_{i_2} \ldots l_{i_k} = r_{i_1} r_{i_2} \ldots r_{i_k}$, and since the tree has exactly one path $q(n_0, n_0)$ is necessarily derived.

Therefore, the empty tuple is inserted to the target predicate $res$, concluding the proof that $G$ is satisfiable.

2.5.2 Identifying all Distinct Values in a Tree

This problem is related to populating the relation $Val$. Given an XML document with $n$ nodes, which we call the tree, we would like to assign each node an additional tag, called value tag, such that two nodes $n_1$ and $n_2$ are assigned the same value tag iff they have the same value. We assume that XML elements’ names and text in XML are strings over an alphabet $\Sigma$. We show that the time
complexity of assigning value tags to an XML document is \( \Theta(n + L) \) where \( L \) is the total length of tags and texts (note that the implemented hash based algorithm for populating \( Val \) gave results linear in \( n \), implying that in the benchmarks, \( L \) is linear in \( n \)). We assume that the value tag of each node can be stored in a single computer word (the value field), that one word is sufficient to store all the pointers in the tree and that a word can also store any value (whose number is less than or equal to the number of pointers).

We use a trie. An array in each of the nodes of the trie has an element for each element in \( \Sigma \) and \( n \) additional elements (for \( n \) integers). Each such integer represents a possible value tag (note that there are at most \( n \) distinct values in the tree). We assume that the \( n \) integers and \( \Sigma \) are disjoint. The initialization of this array is done in \( O(1) \) [8]. Observe that a necessary condition for two nodes to have the same value is that they are roots of subtrees of the same height. We first partition the nodes of the tree to sets of nodes with the same height, by traversing the tree recursively and creating sets with pointers to nodes, such that pointers to the nodes that are roots of subtrees of height \( i \) are in the set \( H_i \) (we denote the number of elements in \( H_i \) by \( n_i \)). The complexity of this procedure is \( O(n) \).

We start by inserting into a trie the strings corresponding to elements in \( H_0 \), i.e., the strings or tags of the leaves. Whenever a new string is inserted into the trie, we assign a value to it, from a counter initialized to zero at the beginning and incremented after each value assignment. This value is written both at the end node of the string in the trie and in the node’s value field. If the inserted string exists in the trie, we read the existing value for the string and store it in the node’s value field. At this point, all the nodes whose pointers are in \( H_0 \) are value tagged.

We then consider the sets \( H_1, H_2 \), and so on, in this order. For every such \( H_i \), we consider for each element \( e \) in \( H_i \) a generalized string, which is the concatenation of \( e \)’s tag and the values (i.e., integers) assigned to \( e \)’s direct children, from left to right. Note that since \( \Sigma \) and the integers used for the values are disjoint, this generalized string corresponds to exactly one value. Also, the trie is designed to handle the generalized strings. Values are assigned from the same counter used for value-tagging the elements in \( H_0 \).

**Upper Bound.** After the partition of nodes into the sets \( H_i \), each node is visited once as the "primary" node whose value is being inserted into the trie, and once again as the child of such a
"primary" node. The visits overall cost $O(n)$ operations. Every text or tag is inserted into the trie once, possibly followed by a sequence of integers. The total number of integers inserted at any point during execution after a text or tag is $n$. In total, $O(n + L)$ operations. Thus, the algorithm operates in $O(n + n + n + L) = O(n + L)$.

**Lower Bound.** Traversing the $n$ nodes in the tree and reading the strings whose total length is $L$ is necessary for giving value tags, as the tagging may well be influenced by the last character read in the last nodes. A lower bound for this problem is therefore $\Omega(n + L)$. In conclusion, the time complexity of assigning value tags to an XML document is $\Theta(n + L)$.

### 2.5.3 $XPath^L$ dialects Expressiveness Results

We define several dialects of $XPath^L$:

1. Core-$XPath^L$, or simply $L$, is $XPath^L$ without the predicates $lValCon$, $eValCon$ and $valSim$.

   We define dialects of Boolean core-$XPath^L$, which are each a subset of the full language as defined below. The dialects are compared, thereby providing insight regarding the contribution of the different language constructs to the language’s expressive power. The Boolean case is rich enough and it exhibits tradeoffs between objects and values and between constant expressions and variables that can be bound to expressions.

2. Core-$XPath^L_v$, or simply $L_v$, the subset of $L$ in which only ids of value objects are allowed to be bound to variables in rule heads. A Core-$XPath^L$ program $p$ is a Core-$XPath^L_v$ program if $p$ meets the syntactic requirement that every variable in a rule’s head appears as the second argument of a $val$ predicate in the rule’s body.

3. Core-$XPath^L_c$, or simply $L_c$, the subset of $L$ in which the second argument of $xpath$ subgoals is always a constant. Note that the relation corresponding to the predicate $xpath$ contains only expressions from $Ed$. The use of expression predicates remains allowed. This dialect is intended to isolate the contribution to the expressive power made possible by allowing expressions to be bound to variables.

**Programs and Execution Trees.** We use the notation and style of [92]. An $XPath^L$ program is the 2-tuple $<rules,target-predicate>$, where $rules$ is a set of rules, and $target-predicate$ is one of
the predicates that appear in a rule head [92]. If the arity of the target predicate is zero, the program is Boolean and returns true iff after evaluation, the empty tuple is in the corresponding relation.

The execution of programs is described via execution trees. These are composed of ground instances (i.e., containing object ids and no variables) of rules. In an execution tree, the rule’s head is a parent of the rule’s subgoals. Formal definitions of various types of execution trees for programs with and without negation can be found in the appendix (they are not needed in order to follow the sketches of proofs below). An execution tree for a program \( P \) exists iff its return value is true.

**Equivalence of Programs.** Boolean programs \( P_1 = \langle r_1, t_1 \rangle \) and \( P_2 = \langle r_2, t_2 \rangle \) are equivalent if for all databases, after evaluation, the relation for IDB predicate \( t_1 \) is empty iff the relation for IDB predicate \( t_2 \) is empty.

**Language containment.** Let \( Q \) and \( P \) be two query languages. We say that \( Q \) contains \( P \) and denote \( P \preceq Q \) if every Boolean query expressible in \( P \) is also expressible in \( Q \). Proper containment, denoted \( P \prec Q \), also implies that there exists a Boolean query expressible in \( Q \) but not in \( P \). If \( P \preceq Q \) and \( Q \preceq P \), we say that \( P \) and \( Q \) are equivalent \((P \equiv Q)\). \( P^+ \) is the subset of language \( P \) in which the use of negation is not allowed.

The first result is \( L \equiv L_v \), which intuitively shows that the constraint of referring to values only in rule heads is completely compensated for, in terms of expressive power, by the ability to bind expressions to variables. The second result is \( L_c \prec L \), which shows that not having the ability to bind expressions to variables reduces the expressive power. The third result is \( L^+ \equiv L_c^+ \), under the assumption that the EDB does not include XPath expressions in tuples of relations or as values of TNOs. This shows that in this setting, binding expressions to variables does not increase expressive power. Next, we sketch the proofs for the results. Full proofs for the results in Sections 2.5.4-2.5.6 appear in the appendix.

### 2.5.4 \( L \equiv L_v \)

By definition, \( L \succeq L_v \). We sketch the proof that \( L_v \succeq L \). We define a transformation which takes a program \( P \) in \( L \) and transforms it into an equivalent program \( P' \) in \( L_v \). For every rule in \( P \) with \( n \) variables, \( P' \) includes \( 2^n \) adorned rules, for every possibility of every variable \( X \) to be bound to a value or to a non-value object. In the former case, \( \text{val}(X, X) \) is added to the rule. In the latter
case, \( \neg val(X, X) \) is added to the rule. The adornment are added to the head (see appendix). The following new rule is included in \( P' \):

\[
\text{bad}(E) \leftarrow \text{root}(W), \text{xpath}(W, E, X), \text{xpath}(W, E, Y), \neg \text{self::*}(X, Y).
\]

Intuitively, an XPath expression is "bad" if its result set has more than one element. We call an expression whose result set is a singleton a **logical pointer**.

Every variable \( X \) whose corresponding adornment in the head is \( l \) (for **logical pointer**) is replaced by a new variable \( E_X \) in both body and head (denoting string whose value is an expression), and the predicates \( \text{root}(R), \text{xpath}(R, E_X, X), \neg \text{bad}(E_X) \) are added to the body, where \( R \) is a new, unused variable. This is done in order to translate from a logical pointer \( (E_X) \) to its corresponding object id (bound to \( X \)), so that references to \( X \) in the rest of the body retain their original semantics.

If the return value of \( P \) is **true**, then \( P \) has an execution tree. Based on this tree, we show that a corresponding execution tree for \( P' \) exists as well. If \( P' \)'s return value is **false**, the transformation shows that an execution tree for \( P' \) cannot exist. Thus, \( P \) is equivalent to \( P' \).

### 2.5.5 \( L_c \prec L \)

By definition, \( L_c \preceq L \). We show a specific example program \( P \) in \( L \), and prove that there is no equivalent program in \( L_c \). The proof assumes the existence of \( P_c \), an equivalent program, and derives a contradiction by showing that \( P_c \) would not give the same return value as \( P \) for at least one database. \( P \) is:

\[
\text{res()} \leftarrow \text{u}(E), \text{u}[*[W,X]], \text{xpath}(X, E, Y), \text{root}(W).
\]

\( P \) returns **true** if there is in \( U \) (the EDB relation corresponding to the predicate \( u(\cdot) \)) a logical pointer which is also in \( E_d \) (see 2.2.4) where \( d \) is the document whose root is in \( \text{Root} \). In a lemma, we prove that for every possible length of expression, there exists an "allowed" expression which is even longer, w.r.t. some document. Let \( q \) be an XML document and let \( e \) be a logical pointer in the set \( E_q \) s.t. \(|e| \) (\( e \)’s length, say in bits) is greater than \(|P_c| \) (\( P_c \)’s length). The intuition is that \( e \) is longer than the whole of \( P_c \) and therefore cannot appear as a constant in its predicates. Since, in addition, \( P_c \) is such that its variables cannot be bound to expressions, every occurrence of \( e_A \) in a corresponding execution tree is a result of a binding to a variable rather than a constant.

\( P \) returns **true** if \( e \) is in relation \( U \). Therefore, \( P_c \) returns **true** as well, and has a hypothetical
execution tree which describes this deduction. The proof shows that if all occurrences of $e$ in the hypothetical tree (which is built of rules whose variables cannot be bound to expressions and that do not contain $e_A$ as a constant) are replaced by the equivalent expression $self::*/e$, which is the concatenation of ”self:*” to the first element in the disjunction which comprises $e$ (see appendix), then the resulting tree implies that $P_e$ returns true when only $self::*/e$ is in $U$. However, by the definition of $E_q$, $self::*/e$ is never in the set $E_q$ for any $q$, a contradiction.

2.5.6 $L^+ \equiv L_e^+$

We assume that there are no XPath expressions in the EDB. By definition, $L^+_e \preceq L^+$. We sketch the proof that $L^+ \preceq L_e^+$. A transformation which takes a program $P$ in $L^+$ and transforms it into a program $P'$ in $L_e^+$ is defined. Let $Z'$ be the set of expressions which explicitly appear in $P$ and also includes the expression $/descendant-or-self::*$, and let $Z$ be the set of the ”allowed” expressions equivalent to the elements in $Z'$. For each rule $r$ in $P$, $P'$ includes a set of rules, one for each possibility to substitute variables which appear in the second argument of an xpath predicate by expressions from $Z$. For example, if $Z' = \{e_1, e_2, /descendant-or-self::*\}$ and a rule $r_0$ has two variables that can be bound to expressions, the transformed program includes $3^2$ rules instead of $r$, one for every possible substitution of e-variables with elements from $Z$.

It is intuitive that if $P'$ returns true on a database, then $P$ returns true as well, since $P'$ represents the substitutions which are needed to deduce $P$’s target predicate.

The proof, which appears in the appendix, also shows that if $P$ returns true, then $P'$ returns true. First, if variables that can be bound to expressions appear only in xpath predicates, they can be substituted by the expression in $Z$ equivalent to $/descendant-or-self::*$. Second, if such variables appear in both xpath and IDB predicates, then in a satisfied rule they must be substituted by an expression which explicitly appears in the program. Note that no other cases exist since in a satisfied rule, such variables cannot appear in an EDB predicate. Therefore, based on $P$’s execution tree, $P'$ also returns true.
2.6 Conjunctive queries with XPath-inspired predicates over DAGs

This section addresses the complexity of evaluating conjunctive queries over labeled directed acyclic graphs (DAGs). DAG-structured data and queries over such data occur in many contexts, such as ontologies for the Semantic Web and Complex Event Processing. The relations representing the DAG are binary axis relations, mostly adopted from XPath, and unary label relations. The relations definitions are generalized for the DAG case. We prove that all polynomial time results of [41], except one, become NP-Complete over DAGs.

2.6.1 Motivation

We describe two motivations for querying DAG-structured data, one related to the area of Semantic Web and the other - to Complex Event Processing.

**Semantic Web.** The Semantic Web is a vision for the future of the World Wide Web according to which the organization of data is according to meaningful concepts. Semantics is expected to dramatically improve the efficiency of knowledge management because data will become more meaningful to machines [98], that will be able to assert the correctness of existing data and extract new knowledge out of existing knowledge [10].

Ontologies are a key concept in the Semantic Web which provide a means to classify and name objects of the modeled domain, their properties and the relationships among them. A crucial step in defining an ontology is the organization of the relevant terms in a taxonomic hierarchy [10], i.e., stating a subclass relationship between classes of objects, which means that if \( X \) is a subclass of \( Y \), then every object in \( X \) is also in \( Y \). A DAG taxonomy is a taxonomy where a class may be a subclass of one class or more, as opposed to a tree taxonomy, where a class can be a subclass of at most one class. The Resource Description Framework Schema (RDFS) [97] is a framework for constructing ontologies. Classes in RDFS ontologies are organized in a DAG taxonomy (cycles imply that participating classes are equivalent and can be eliminated from the graph). A Class may be assigned properties explicitly or inherit properties from a superclass. Properties themselves may also be organized as a DAG taxonomy. The information in the ontology provides semantics to data in Resource Description Framework (RDF) [97] instances which refer that ontology. This is done
by RDF elements *subjects*, *objects* and *predicates* referring to RDFS elements as types.

We use an example to demonstrate the use of conjunctive queries over DAGs in the Semantic Web context. Figure 2.19(a) depicts an ontology for an electronic bookstore. The convention we use to represent the ontology as a graph is taken from [10] with a slight technical modification. An ellipse represents a class and a rectangle represents a property. A directed edge $$(u,v)$$ where $$u$$ and $$v$$ are both classes (respectively, properties) means that $$u$$ is a subclass (respectively, subproperty) of $$v$$. While in [10] edges labeled by $$d$$ (for domain) and $$r$$ (for range) connect a property to its domain and range classes of values, we simulate this edge label by splitting the edge into two edges and adding a new node, labeled either $$d$$ or $$r$$. One edge connects the property to the new node and the other edge connects the new node to the class. Therefore, edges in our graph are label-free.

The ontology in Figure 2.19(a) specifies the following taxonomy: Chemistry Book, Physics Book and Biology Book are subclasses of Scientific Book. The latter and Novel are each a subclass of Book. Person is a superclass of Scientist and of Artist. The figure does not show that all classes are subclasses of Class, which is the class of all classes in RDFS.

For example, Chemistry Book inherits the properties ”Scientific Author”, ”Author” and ”Title” from its superclasses. The following conjunctive query (whose conjuncts’ semantics are to be formally defined later on) extracts from the graph all the classes in the ontology that are in the range of the property ”Author”, explicitly or by inheritance.

$$Res(X) \leftarrow Child^*(X,W), Child(P,R), Label_{Author}(P), r(R), Child(R,W), Class(X).$$ Note that arrows between classes are in the leaf to root direction. After the evaluation of this conjunctive query, the
relation $Res(\cdot)$ will contain all the classes in the ontology that may be in the range of the property "Author" in an RDF instance which refers to the above RDFS ($Class(\cdot)$ denotes a relation whose elements are all the classes).

**Causal Event Histories.** The area of Complex Event Processing (CEP) deals with tracking the causality between events, i.e., which event(s) caused additional complex events (also known as "situations" [5]) to occur. CEP has many applications ranging from low level infrastructure management to high level business intelligence. The log of the history of events can be represented as a DAG in which every node represents an instance of one event type, and an edge $(u,v)$ exists if $u$ caused $v$ (solely or together with other events) [63]. This DAG does not represent rules that define triggering relationships between event types, but represents what actually happened in the CEP system in terms of instances of events. Conjunctive queries over DAGs can be a powerful root-cause analysis tool and a means to gather information about the modeled reality. For example, the following Boolean query checks if the pattern in Figure 2.19(b) exists in a causal event history.

$$Q() \leftarrow e1(X), Child(X,Y), e4(Y), Child^*(X,Z), e2(Z), Child(Y,W), Child(Z,W), e3(W).$$

The pattern means that event $e1$ caused $e4$ directly. $e1$ also caused a possibly empty sequence of events followed by an $e2$ event. Events $e2$ and $e4$ together caused $e3$.

### 2.6.2 Complexity of Conjunctive Queries over Trees

Gottlob and Schulz [41] characterize the complexity of conjunctive queries over trees. The relations which represent the data are axis-relations adopted from XPath axes and label relations over a labeling alphabet $\Sigma$. A *signature* is a set of relations, and a conjunctive query over a signature $s$ is a conjunctive query whose conjuncts are taken only from $s$. The following table summarizes the complexity results:

<table>
<thead>
<tr>
<th>Signature</th>
<th>Complexity Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>${Child^*(\cdot, \cdot), Child^+(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}$</td>
<td>$P$</td>
</tr>
<tr>
<td>${Following(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}$</td>
<td>$P$</td>
</tr>
<tr>
<td>${Child(\cdot, \cdot), NextSibling(\cdot, \cdot), NextSibling^+(\cdot, \cdot),$</td>
<td>$P$</td>
</tr>
<tr>
<td>$NextSibling^*(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}$</td>
<td></td>
</tr>
<tr>
<td>Every signature not included in one of the above signatures</td>
<td>NP-Complete</td>
</tr>
</tbody>
</table>
Polynomial results are w.r.t. combined complexity [1, 93] and NP-Completeness results are w.r.t. query complexity (i.e., expression complexity) [1, 93].

### 2.6.3 Results

We generalize the definition of axis relations, originating in XPath queries over trees, to the DAG case. We characterize the complexity of conjunctive queries over DAGs. Our results appear in the following table.

<table>
<thead>
<tr>
<th>Signature</th>
<th>Complexity – DAG</th>
<th>Complexity – Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>{Child^+(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}</td>
<td>NP-Complete</td>
<td>P</td>
</tr>
<tr>
<td>{Child^*(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}</td>
<td>NP-Complete</td>
<td>P</td>
</tr>
<tr>
<td>{Child(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}</td>
<td>NP-Complete</td>
<td>P</td>
</tr>
<tr>
<td>{Following(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{NextSibling(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}</td>
<td>NP-Complete</td>
<td>P</td>
</tr>
<tr>
<td>{NextSibling^+(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}</td>
<td>NP-Complete</td>
<td>P</td>
</tr>
<tr>
<td>{NextSibling^*(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}}</td>
<td>NP-Complete</td>
<td>P</td>
</tr>
</tbody>
</table>

Here too, polynomial results are w.r.t. combined complexity and NP-Completeness results are w.r.t. query complexity [1, 93].

### 2.6.4 Definitions

We represent the data using relations as follows. The relations representing the structure of the DAG, originating from XPath, are adopted from [41]:

- \(Child(\cdot, \cdot)\),
- \(Child^+(\cdot, \cdot)\),
- \(Child^*(\cdot, \cdot)\),
- \(Following(\cdot, \cdot)\),
- \(NextSibling^+(\cdot, \cdot)\),
- \(NextSibling(\cdot, \cdot)\),
- \(NextSibling^*(\cdot, \cdot)\).

When discussing trees, the semantics of the first five relations is identical to the semantics of
the corresponding XPath [105] axes \{\text{child, descendant, descendant-or-self, following, following-sibling}\} respectively.

NextSibling(\cdot, \cdot) corresponds to following-sibling[position()=1] and \((x,y) \in \text{NextSibling}^{\cdot}(\cdot, \cdot)\) iff \((x,y) \in \text{NextSibling}^{+}(\cdot, \cdot) \cup \{(x,x)\} \).  

Note that this is a superset of XPath, since every axis is either in the list of predicates above or implicitly covered by its reverse axis. For example, a relation Parent(\cdot, \cdot) with semantics corresponding to the XPath axis "parent" can be simulated by the Child(\cdot, \cdot) relations with swapped arguments.

### 2.6.4.1 Generalization to DAGs.

In the case of DAGs, these relations remain a natural way to express conjunctive queries, but a generalization of the definition for some of them is required. Following are definitions of the axis relations, which represent the DAG structure. We denote by \((V,E)\) the directed graph corresponding to the DAG expressed by the relations. Since we discuss ordered graphs, The schema of \(E\) is \(E(X,Y,N)\) and elements are of the form \((x,y,n)\) where \(x\) is the source of the edge, \(y\) is the destination and \(n\) is the number of \(y\)’s position among \(x\)’s outgoing edges in the order of the children’s object IDs, a unique value that exists for every object.

- \((x,y) \in \text{Child}(\cdot, \cdot) \iff \exists n \text{ such that } (x,y,n) \in E.\)

- \((x,y) \in \text{Child}^{\cdot}(\cdot, \cdot) \iff (x,y) \in (\pi_{X,Y} E)^{\cdot}, \text{ where } R^{\cdot} \text{ denotes the reflexive and transitive closure of the relation } R \text{ (i.e., reachable via edges in zero or more steps)}.\)

- \((x,y) \in \text{Child}^{+}(\cdot, \cdot) \iff (x,y) \in (\pi_{X,Y} E)^{+}, \text{ where } R^{+} \text{ denotes the transitive closure of the relation } R \text{ (i.e., reachable via edges in one or more steps)}.\)

- \((x,y) \in \text{Following}(\cdot, \cdot) \iff x \text{ precedes } y \text{ in the postorder [92] numbering where the children of a node are explored according to their position relative to their siblings, and } (y,x) \notin \text{Child}^{\cdot}(\cdot, \cdot). \text{ If there is more than one root, roots are explored in the order of their object IDs.}\)

- \((x,y) \in \text{NextSibling}^{\cdot}(\cdot, \cdot) \iff \exists v \in V \text{s.t.} ((v,x,n) \in E \land (v,y,m) \in E \land (m > n)).\)
• \((x, y) \in NextSibling^*(\cdot, \cdot) \iff \exists v \in V s.t. ((v, x, n) \in E \land (v, y, m) \in E \land (m \geq n))\).

• \((x, y) \in NextSibling(\cdot, \cdot) \iff \exists v \in V s.t. ((v, x, n) \in E \land (v, y, n + 1) \in E)\).

Nodes are labeled. We denote the labeling alphabet by \(\Sigma\), and allow multiple labels for the sake of simplicity of proofs. The tractability result remains valid for singly labeled graphs. After proving NP-Hardness, we show how multiple labels can be eliminated from each of the proofs.

\[2.6.5\] Complexity of conjunctive queries over trees

\[2.6.5.1\] Polynomial Results.

We briefly summarize the methodology used in [41] to prove polynomial time results for trees as it is used for the DAG case. Let \(R\) be a binary relation in a relational structure \(S\) and \(<\) be a total order on \(S\)'s domain, \(A\). Then, we call \(R <\)-hemichordal if for all \(n_0, n_1, n_2, n_3 \in A\) s.t. \(n_0 < n_1\) and \(n_0 \leq n_2 \leq n_3\),

\[R(n_1, n_2) \land R(n_0, n_3) \rightarrow R(n_0, n_2)\] and \(R(n_2, n_1) \land R(n_3, n_0) \rightarrow R(n_2, n_0)\).

**Lemma 2.4** Let \(A\) be the finite domain of values for a relational structure \(S\), \(R\) be a binary relation on \(S\) and \(<\) be a total order on \(A\) s.t. \(R \subseteq \leq\). Then, \(R\) is \(<\)-hemichordal iff for all \(n_0, n_1, n_2, n_3 \in A\) s.t. \(n_0 < n_1 \leq n_2 < n_3\),

\[R(n_1, n_2) \land R(n_0, n_3) \rightarrow R(n_0, n_2)\].

Proof and illustrations for the definitions may be found in [41]. Lemma 2.4 provides a means to prove hemichordality in the case where the relation is a subset of the total order with respect to which one tries to prove hemichordality.

**Lemma 2.5** Let \(S\) be a relational structure of unary and binary relations. The evaluation of a Boolean conjunctive query \(Q\) over a structure \(S\) in which all binary relations are \(<\)-hemichordal has polynomial time complexity w.r.t. combined complexity.

The evaluation of a \(k\)-ary conjunctive query (a query with \(k\) variables in the query’s head) can be done by evaluating \(|A|^k\) Boolean conjunctive queries, where \(|A|\) is the size of the finite domain of values for the structure under discussion. Therefore, tractability results which hold for Boolean conjunctive queries, hold for \(k\)-ary conjunctive queries as well, for a fixed \(k\). Proofs for the claims
in Section 2.6.5 can be found in [41], which uses them in order to prove the polynomial results in 2.6.2 by showing that each of the three signatures in 2.6.2 (P) contains unary relations and binary relations which are \textit{-hemichordal} w.r.t. the same total order <.

### 2.6.5.2 NP-Completeness Results

We briefly show the reduction used to prove the NP-Completeness of conjunctive queries over trees over the signature \( \{ \text{Child}(\cdot, \cdot), \text{Child}^+(\cdot, \cdot), \text{Label}_a(\cdot)_{a \in \Sigma} \} \), as we use this later on, by reducing it in order to prove NP-Hardness of the problems under discussion in this paper. 1\text{-in-3} 3SAT\textsuperscript{+} [40] is the NP-Complete problem of determining if there exists a truth assignment for the variables of a 3SAT instance \( I \) with only positive literals such that every clause in \( I \) has exactly one true literal out of its three different literals. Denote \( I = C_1, ..., C_m \) where \( C_i \) represents the disjunction of three positive literals. The reduction polynomially builds a conjunctive query based on this instance, which evaluates to \textbf{true} on the fixed tree in Figure 2.20 iff there is a satisfying truth assignment for \( I \).

In the tree in Figure 2.20, every node appears with an ID and some of the nodes appear with one or more labels. Labels are capitalized, IDs are not. For example, the node with ID \( \text{w}_{3,9} \) is labeled by \( L1 \) and \( L2 \). Edges represent the \textit{Child}(\cdot, \cdot) relation between nodes. The query is built as
follows. For $1 \leq i \leq m$, the new variables $x_i$ and $y_i$ are introduced. Whenever the $k$-th literal of $C_i$ coincides with the $l$-th literal of $C_j$ ($1 \leq i, j \leq m$, $1 \leq k, l \leq 3$, $i \neq j$), a new variable $z_{k,l,i,j}$ is introduced. The query’s conjuncts are:

- For $1 \leq i \leq m$, the conjuncts $C(x_i)$, $B(y_i)$, $Child^3(x_i, y_i)$ are included, where for $k > 1$, $A^k(s_0, e_0)$ stands for $A(s_0, n_0), A(n_0, n_1), \ldots, A(n_{k-2}, e_0)$ where the variables $n_i$ are new and distinct variables.

- For each variable $z_{k,l,i,j}$, the three conjuncts $L_k(z_{k,l,i,j})$, $Child^+(y_i, z_{k,l,i,j})$ and $Child^8+k-l(x_i, z_{k,l,i,j})$ are included.

The basic idea used in our proofs is to modify the tree into a DAG (by deleting original edges and adding new nodes and edges) and provide a new Boolean query that evaluates to true on the DAG iff the above query evaluates to true on the tree.

### 2.6.6 Complexity Results for Conjunctive Queries over DAGs

#### 2.6.6.1 Polynomial Results

Consider the postorder numbering of the nodes in the DAG. If more than one root exists, roots are explored in the order of their object IDs (i.e., as if they had one common parent). Let $<$ be the total order corresponding to this postorder numbering. It is easy to see that $\{ (x,y) \text{ s.t. } Following(x,y) \} \subseteq \{ (x,y) \text{ s.t. } x < y \}$.

**Lemma 2.6** Following$(\cdot, \cdot)$ is $<$-hemichordal.

**Proof:** Following$(\cdot, \cdot) \subseteq <$, therefore we can use Lemma 2.5. Assume that $n_0, n_1, n_2, n_3$ are nodes in the DAG such that $n_0 < n_1 < n_2 < n_3$, and that Following$(n_1, n_2)$ and Following$(n_0, n_3)$. $n_0 < n_1$ entails exactly one of the following:

- $Child^+(n_1, n_0)$, which together with Following$(n_1, n_2)$ entails Following$(n_0, n_2)$ by definition.

- Following$(n_0, n_1)$, which together with Following$(n_1, n_2)$ entails Following$(n_0, n_2)$ since Following$(\cdot, \cdot)$ is transitive.
No other options exist since $<$ equals to the union of the inverse of $Child^+(\cdot, \cdot)$ and $Following(\cdot, \cdot)$. This proof is very similar to the proof of $\sim$-hemichordality of $Following(\cdot, \cdot)$ in [41], however our definition of the relation $Following(\cdot, \cdot)$ is different in the DAG case.

**Theorem 2.7** Conjunctive queries of arity $k$ over the signature \{Following(\cdot, \cdot), Label_a(\cdot)_{a \in \Sigma}\} are in P w.r.t. combined complexity.

**Proof:** According to Lemma 2.6, $Following(\cdot, \cdot)$ is $\sim$-hemichordal. Therefore, according to Lemma 2.5, we conclude that the problem is in P.

### 2.6.6.2 NP-Completeness Results

**NP.**

**Theorem 2.8** The problem of evaluating a Boolean conjunctive query over a DAG, which may use all the axis or label relations, is in NP w.r.t. query complexity (i.e., with a fixed database).

**Proof:** Consider a conjunctive query $q$ having $\text{Var}(q)$ distinct variables in it. A non-deterministic Turing machine can guess a node for each variable, and check for each conjunct whether or not it evaluates to true under that assignment. Guessing the nodes requires polynomial time. We now show that checking the truth value for each conjunct can be done in polynomial time.

- $Child(x_0, y_0)$. Checking if $x_0$ is one of the parents of $y_0$ is $O(|E|)$ operations, simply by checking all of $x_0$’s children.

- $Child^*(x_0, y_0)$. Checking if $x_0$ is one of the ancestors of $y_0$ is $O(|E|)$ operations, by preforming a depth-first search from $x_0$.

- $Child^+(x_0, y_0)$. Same as the previous case, plus checking in constant time that $x_0$ and $y_0$ are not the same node.

- $Following(x_0, y_0)$. Performing depth-first search on the DAG and checking if $x_0$ is visited before $y_0$, but is not one of $y_0$’s ancestors costs $O(|V| + |E|)$ operations.

- $NextSibling(x_0, y_0)$. Checking if $x_0$ is the previous sibling of $y_0$ is also $O(|E|)$, by checking for each of the parents whether $y_0$ is $x_0$’s next sibling relative to that parent.
Figure 2.21: DAG for proving NP-Hardness of \( \{ \text{Child}^+(\cdot, \cdot), \text{Label}_a(\cdot)_{a \in \Sigma} \} \).

- **NextSibling\(^+\)(x_0, y_0).** Same as the previous case, without the restriction that there is no other node after \( x_0 \) and before \( y_0 \).

- **NextSibling\(^*\)(x_0, y_0).** Same as the previous case, but here \( x_0 = y_0 \) may hold.

There is a satisfying assignment of nodes to variables iff the non-deterministic Turing machine returns a positive answer in polynomial time. Therefore, the problem is in NP.

**NP-Hardness.**

**Theorem 2.9** Conjunctive queries over the signature \( \{ \text{Child}^+(\cdot, \cdot), \text{Label}_a(\cdot)_{a \in \Sigma} \} \) are NP-Hard (and therefore NP-Complete) w.r.t. query complexity.

**Proof:** We show a reduction from the NP-Hard problem of Boolean conjunctive queries over trees which uses the signature \( \{ \text{Child}^+(\cdot, \cdot), \text{Child}(\cdot, \cdot), \text{Label}_a(\cdot)_{a \in \Sigma} \} \) to the problem of Boolean conjunctive queries over DAGs which use the signature \( \{ \text{Child}^+(\cdot, \cdot), \text{Label}_a(\cdot)_{a \in \Sigma} \} \). We take the tree of Figure 2.20 and transform it to form the DAG in Figure 2.21. We also rewrite the query, polynomially, so it would use conjuncts from the signature \( \{ \text{Child}^+(\cdot, \cdot), \text{Label}_a(\cdot)_{a \in \Sigma} \} \) in a way which ensures that the rewritten query is satisfied by the DAG iff the original query is satisfied by the tree which completes the proof.

Consider the DAG in Figure 2.21. It consists of all the nodes of the tree in Figure 2.20. All the original edges were deleted. Following is a list of nodes and edges that are added. Note that in order not to overload Figure 2.21, only samples of the new nodes and edges appear in it:
• An edge from every $C$-labeled node to the $B$-labeled node that used to be of distance three edges from the $C$-labeled node in the original tree.

• An edge from every $B$-labeled node to a new $D$-labeled node is added (three new $D$-labeled nodes are added, only one is shown in Figure 2.21).

• Edges from each $D$-labeled node $d$ to all the $L$-labeled nodes that used to be descendants of $d$’s parent (a $B$-labeled node).

• Edges from every $C$-labeled node to a new node labeled by $E^8_k - l$ where $1 \leq k, l \leq 3$, and from every $E^8_k - l$-labeled node to the three $L$-labeled nodes that used to be of distance $8+k-l$ edges from the $C$-labeled node in the original tree.

The query for this DAG, denoted $q'$, is a modification of $q$, the query that was used for the tree, and is built as follows:

• For $1 \leq i \leq m$, the conjuncts $C(x_i)$ and $B(y_i)$ remain the same and $\text{Child}^+(x_i, y_i)$ replaces $\text{Child}^3(x_i, y_i)$.

• The conjunct $L_k(z_{k,l,i,j})$ remains the same.

• The conjuncts $\text{Child}^+(y_i, d), D(d), \text{Child}^+(d, z_{k,l,i,j})$ replace $\text{Child}^+(y_i, z_{k,l,i,j})$.

• The conjuncts $\text{Child}^+(x_j, e), E^8_{k-l}(e), \text{Child}^+(e, z_{k,l,i,j})$ replace $\text{Child}^{8+k-l}(x_j, z_{k,l,i,j})$.

Note that $q'$ uses the $\text{Child}^+(\cdot, \cdot)$ relation only, and is therefore compatible with the signature $\{\text{Child}^+(\cdot, \cdot), \text{Label}_a(\cdot)_{a \in \Sigma}\}$. The procedure of creating the query is polynomial, since every conjunct either stays the same or is replaced by a constant number of conjuncts. We argue that $q$ is satisfied when evaluated on the original tree iff $q'$ is satisfied when evaluated on the DAG.

• In the DAG every $C$-labeled node has exactly one $B$-labeled descendant. Evaluating $q$ on the tree, $\text{Child}^3(x_i, y_i)$ forces the bindings for $\{x, y\}$ to only be $\{v_1, w_{1,1}\}$ or $\{v_2, w_{2,2}\}$ or $\{v_3, w_{3,3}\}$. Evaluating $q'$ on the DAG, $\text{Child}^+(x_i, y_i)$ forces the same bindings. Since the conjuncts $C(x_i)$ and $B(y_i)$ remain the same, and none of the new nodes are $B$ or $C$ labeled, $x_i$ and $y_i$ have the exact same possible bindings in both cases.
• $z_{k,l,i,j}$ can only be bound to the one node which is both a descendant of distance $8 + k - l$ (in edges) from $x_j$ and a descendant of $y_i$, because of the conjuncts $L_k(z_{k,l,i,j})$, $Child^+(y_i,z_{k,l,i,j})$ and $Child^{8+k-l}(x_j,z_{k,l,i,j})$ in $q$. In the DAG, the semantics of $Child^+(y_i,z_{k,l,i,j})$ is achieved by $Child^+(y_i,d), D(d), Child^+(d,z_{k,l,i,j})$ since nodes that used to be descendants of a $B$-labeled node in the tree ($y_i$) are now children of the corresponding $D$-labeled node. The conjuncts $Child^+(x_j,e), E^{8+k-l}(e), Child^+(e,z_{k,l,i,j})$ force that the only possibility for a binding for $z_{k,l,i,j}$ would be the node which used to be of distance $8+k-l$ edges distant $x_j$, exactly as it is with $q$ applied to the tree.

This completes the reduction, which shows that the problem under discussion is NP-Hard. According to Theorem 2.8 the problem is in NP, therefore, the problem is NP-Complete.

**Theorem 2.10** Conjunctive queries over the signature $\{\text{Child}(\cdot,\cdot), \text{Label}_a(\cdot)_{a \in \Sigma}\}$ are NP-Complete w.r.t. query complexity.

**Proof:** The proof for $\{\text{Child}^+(\cdot,\cdot), \text{Label}_a(\cdot)_{a \in \Sigma}\}$ holds here as well, after replacing, in the query, each of the $Child^+(\cdot,\cdot)$ conjuncts with a $Child(\cdot,\cdot)$ conjunct. The query forces that a satisfying binding for a $Child^+(\cdot,\cdot)$ occurrence also satisfies $Child(\cdot,\cdot)$.

**Theorem 2.11** Conjunctive queries over the signature $\{\text{Child}^*(\cdot,\cdot), \text{Label}_a(\cdot)_{a \in \Sigma}\}$ are NP-Complete w.r.t. query complexity.

**Proof:** The proof for $\{\text{Child}^+(\cdot,\cdot), \text{Label}_a(\cdot)_{a \in \Sigma}\}$ holds here as well, after replacing all $Child^+(\cdot,\cdot)$ conjuncts with $Child^*(\cdot,\cdot)$ conjuncts, since the labels in the query force that $Child^*(x_0,x_0)$ does not hold for any $x_0$.

**Theorem 2.12** Conjunctive queries over the signature $\{\text{NextSibling}(\cdot,\cdot), \text{Label}_a(\cdot)_{a \in \Sigma}\}$ are NP-Complete w.r.t. query complexity.

**Proof:** We show a reduction from the NP-Complete problem of Boolean conjunctive queries over DAGs which use the signature $\{\text{Child}^+(\cdot,\cdot), \text{Label}_a(\cdot)_{a \in \Sigma}\}$ to the problem of Boolean conjunctive queries over DAGs which use the signature $\{\text{NextSibling}(\cdot,\cdot), \text{Label}_a(\cdot)_{a \in \Sigma}\}$. We take the DAG described in Figure 2.21 and transform it to form the DAG in Figure 2.22. We also rewrite the query, polynomially, so it would use conjuncts from the signature $\{\text{NextSibling}(\cdot,\cdot)\}$. 

80
Figure 2.22: DAG for proving NP-Hardness of \{\text{NextSibling}(\cdot,\cdot),\text{Label}_a(\cdot)_{a\in\Sigma}\}.

\text{Label}_a(\cdot)_{a\in\Sigma}\} in a way which ensures that the rewritten query is satisfied by the DAG of Figure 2.22 iff the original query is satisfied by the DAG of Figure 2.22, similar to the way we proved in Theorem 2.9.

The DAG used in proving Theorem 2.9 (see Figure 2.21) is changed as follows. All the edges are deleted, and a new node is added for each deleted edge. Denote by \(n(u,v)\) the node added when deleting the edge \((u,v,k)\). The edges \((n(u,v),u,1)\) and \((n(u,v),v,2)\) are added. The query is modified by replacing every \(\text{Child}^+(\cdot,\cdot)\) conjunct by a \(\text{NextSibling}(\cdot,\cdot)\) with the same parameters. The semantics in terms of node bindings are kept identical, since every node that used to be a child of another node \(n\) in the original DAG is now \(n\)'s next sibling with respect to the added parent node.

The rewritten query is satisfied by the DAG in Figure 2.22 iff the original query is satisfied by the DAG from Figure 2.21 since node bindings to variables are identical. Therefore, the problem is NP-Hard. According to Theorem 2.8 the problem is in NP, therefore, the problem is NP-Complete.

**Theorem 2.13** Conjunctive queries over the signature \{\text{NextSibling}^+(\cdot,\cdot),\text{Label}_a(\cdot)_{a\in\Sigma}\} are NP-Complete w.r.t. query complexity.

**Proof:** No node has more than two children, therefore \(\text{NextSibling}^+(\cdot,\cdot)\) and \(\text{NextSibling}(\cdot,\cdot)\) coincide on the DAG of Figure 2.22. Replacing \(\text{NextSibling}(\cdot,\cdot)\) conjuncts with \(\text{NextSibling}^+(\cdot,\cdot)\) conjuncts with the same parameters will have no effect on bindings, therefore the rewritten query is satisfied iff the query of Theorem 2.12 is satisfied. Therefore, the problem is NP-Hard. According to Theorem 2.8 the problem is in NP, therefore, the problem is NP-Complete.
Theorem 2.14 Conjointive queries over the signature \{NextSibling\(^\ast(\cdot,\cdot), Label_a(\cdot)_{a \in \Sigma}\}\) are NP-Complete w.r.t. query complexity.

Proof: The proof for \{NextSibling\(+(\cdot,\cdot), Label_a(\cdot)_{a \in \Sigma}\}\} (Theorem 2.13) holds for this case as well, after replacing all NextSibling\(+(\cdot,\cdot)\) conjuncts with NextSibling\(^\ast(\cdot,\cdot)\) conjuncts, Since the labels in the query force that NextSibling\(^\ast(x_0,x_0)\) does not hold for any \(x_0\).

2.6.6.3 Elimination of multiple labels.

For the proofs of Theorems 2.9, 2.10 and 2.11, eliminating multiple labels can be done by adding to every \(L_k\)-labeled node \(l\) up to three children, each with a different single label, and label \(l\) with \(G\). In the query, delete \(L_k(z_{k,l,i,j})\), and add \(G(z_{k,l,i,j}), L_k(p), Child^\ast(z_{k,l,i,j}, p)\), where for Theorem 2.11 \(\bullet\) is \(*\), for Theorem 2.9 \(\bullet\) is \(+\), and it is empty for Theorem 2.10.

For the proofs of Theorems 2.12, 2.13 and 2.14, eliminating multiple labels can be done by adding to every \(L_k\)-labeled node \(l\) up to three siblings, each through a new, different parent, and label \(l\) with \(G\). In the query, delete \(L_k(z_{k,l,i,j})\), and add \(G(z_{k,l,i,j}), L_k(p), NextSibling^\ast(z_{k,l,i,j}, p)\), where for Theorem 2.14, \(\bullet\) is \(*\), for Theorem 2.13, \(\bullet\) is \(+\), and it is empty for Theorem 2.12.

2.7 Related Work

Several logic-based languages for querying semi-structured data have been proposed. For example, \textit{E-log} [14] is an HTML-oriented Datalog-like language for information extraction from HTML documents (which also includes a feature of inter-document crawling).

\textit{XPath-Logic} [64] is an XPath extension, inspired by \textit{F-logic} [60]. Similarly to XPath, it provides navigational access to nodes, but, in addition, allows recursion and variable binding for querying and manipulating XML documents. It does not however integrate relational and XML data and does not process values.

\textit{StruQL} [37] operates over databases that are edge-labeled directed graphs. A StruQL query can use regular expressions to select nodes from the graph and builds new graph(s) without changing
the original graph. However, it neither uses XPath nor integrates relational and XML data.

**SQL4X** [26] is an SQL-based language for querying both relational and XML databases. SQL4X queries do not use XPath and do not have iteration or recursion. Results can be relations or XML documents. Datalog$^4_X$ is a language for conjunctive queries which models SQL4X queries.

**SQL/XML** [88] is a standard for integrating XML processing with SQL, which defines integration-related functions and XQuery processing (using the function $XMLQUERY$) over a document or a fragment. It is neither logic-based nor object oriented, but rather an extension to SQL. Note that the function $extractValue$ in Oracle’s SQL/XML extracts the literal value (rather than a node set) of an XPath expression. In $XPath_L^L$, the term $value$, with a different semantics, is defined in Section 2.2.2.5.

**XCerpt** [16] is a rule-based language for querying semi-structured data with recursion capabilities. XML-$QL$ [31] uses pattern matching and regular expressions to query XML data. XCerpt and XML-$QL$ do not integrate relational and XML data.

**System RX** [17] is a hybrid relational and XML database system. System RX leverages the knowledge, and software base, in relational processing for XML and integrative XML and relational processing. It uses SQL/XML and XQuery. Similarly to the approach presented in this paper, XML data is parsed rather than stored as BLOBs or CLOBs. In System RX, only the root of an XML document or fragment can be pointed from a tuple’s column, while in our model any node can be referenced. System RX uses the DB2 optimizer, with modifications, for XML processing, while in the runtime system that we constructed, there are two distinct query processors (and possibly optimizers) with conjunctive-query optimization on top. Our approach generalizes to an arbitrary number of query processors.

The DB2 version 9 system [27] offers XML processing features which are based on the ability to store a reference to an XML document (i.e., to its root) as a field in a tuple. This system also provides rich indexing capabilities. Similar advanced relational and XML processing capabilities
are also provided by Oracle XML DB products [68] and in MS-SQL [69]. In our data model, every XML node id (not necessarily just the root) can be stored in tuple fields. Although not discussed in this paper, various indexes can also be added.

Duplicate-Elimination in relational database was studied extensively. In XPath, elimination is based on node identity, while in SQL, elimination is value-based. In [61], simulating identity-based elimination in SQL is considered. In this work, we considered and implemented the notion of nodes with different identity but with the same value nodes (isomorphic subtrees).

The complexity of evaluating CQs over tree-structured data with XPath-inspired predicates is studied in [42]. Algorithms for evaluating conjunctive queries over tree-structured and graph-structured data (as a model for various query languages for semi-structured data) are studied in [21].

Many of the results in this chapter were published in [78], [79]. Specifically, the results on complexity of evaluating CQs on DAG-structured data were published in [77].
Chapter 3

Query Networks

3.1 Introduction

Web 2.0 is a general term for internet applications which interconnect their end users either by providing a connectivity platform or by enabling users to share contents with each other (or both). Some examples for Web 2.0 applications are blog-hosting sites, wikis, video-sharing sites, dating sites, instant messaging applications and social networks. For example, Wikipedia [100] is an encyclopedia whose entries are authored, edited and maintained by the users, as opposed to, e.g., the Britannica online encyclopedia [20] which simply provides users with contents.

The most successful type of Web 2.0 application is the Social Network. Some examples of social networks are Facebook [35], MySpace [70] and LinkedIn [62]. Social networking applications also exist within corporations, universities and other organizations. Typically, a participant in a social network is associated with some information (such as name, photograph, interests), and with a list of connections to other participants. Participants can interact and share contents using various features.

Besides being a useful platform for social interactions, social networks are increasingly being used as a tool for business development and management. LinkedIn [62], for example, is a social network dedicated to professional networking. The usefulness and popularity of social networks results in increasing amounts of data that are available to their users. Such amounts of data are hard and expensive to manage manually. Indeed, in Facebook, users can take advantage of a simple query language [36] in order to process, mostly, their own and their immediate friends’
data. However, the main social networking feature of managing connections between participants is still managed manually.

In addition to being tools to manage increasing volumes of data using queries, we believe that automatic data management features for social networks will be required in order to (conceptually) provide interaction between users even when they are not online. Ultimately, these features will evolve to become agents which represent, and act on behalf of, the participants in the network.

In this chapter, we consider a near future scenario in which participants of a social network manage their data automatically. In particular, we examine the probable case in which participants define, in a form of a query, with whom they would like to be connected.

From an algorithmic point of view, evaluation of these queries has to be done on a large scale, an issue which raises many practical and theoretical issues addressed throughout this chapter.

### 3.1.1 Motivational Example

Consider the social network illustrated in Figure 3.1. Seven participants are shown with their current connections in the network (the connections are represented by directed edges, an issue discussed in Section 3.1.2). The network participants have policies which define which connections they would like to add to themselves:

- Lisa would like to connect to every participant who is a friend of two distinct friends of hers.
- Bart, on the other hand, would like to connect to every participant who is connected to him through two distinct paths, such that one path is of length 2 edges and the other - of length 3.

These two definitions of policies are formally given later on as queries $q_a$ (Lisa) and $q_b$ (Bart) in Section 3.2, and are illustrated graphically in Figure 3.4. However, the formal definitions are not needed in the context of this general example. The rest of the participants are associated with queries as shown in Figure 3.1.

Lisa is not connected to Marge. However, Lisa is connected to Homer and to Pluto, who are both connected to Marge.

- According to Lisa’s query, we add the edge $(Lisa, Marge)$ to the network. In Figure 3.2, this edge is marked by 1.
- Based on this added edge, Maggie became a participant to whom Bart is connected through two
distinct paths. A two-edge path through Mickey, and a three-edge path through Lisa and Marge. We therefore add the edge \((Bart, Maggie)\) to the network (marked 2 in Figure 3.2).

- Note that the addition of edge 2 is done based on edge 1 and on original network edges. The addition of edge 2 renders Maggie, who is not connected to Lisa, as a participant who is connected to two distinct friends of Lisa. The edge \((Lisa, Maggie)\), marked 3 in Figure 3.2 is therefore added. Note that edge 3 is added based on original edges and edge 2.

The addition of edges in this example demonstrates the recursive nature of adding edges to the network. Lisa’s query was evaluated and edge 1 was added. Based on this addition (but not based on it only), another evaluation of Lisa’s query resulted in the addition of edge 3. As we will shortly see, we model this recursion using the Datalog formalism.

### 3.1.2 Model

Our model is a graph-based formalism, in which nodes represent the network participants, and edges represent the connections between them. The model does not assume connection reciprocity. That is, it is possible that participant \(a\) lists \(b\) as a friend, but \(b\) does not list \(a\). Directed edges are therefore used to model connections. Some social networks, such as Twitter, have directed connections between participants. Others, such as Facebook, typically have undirected connections between participants. We model connections using directed edges. Directed edges are capable of
modeling both with and without reciprocity. In addition, the directed graph makes the definition of queries and their semantics more succinct.

Also, we believe that the fast development of social networks and their proliferation to business and organizational cultures will result in new types of connections whose modeling may require the more general model, with directed edges. For example, connections between fans and a rock star or between pupils and their teacher.

Every node, say $n$, has a query associated with it. This query defines the nodes that the participant corresponding to $n$ would like to add to her friends’ list. As in most social interactions, the query will make use of existing connections in order to create new ones.

**Conjunctive Queries.** Our query formalization is that of a *Conjunctive Query* (CQ), which we describe in Chapter 2 (also see [92]). The choice of the CQ model is motivated by the same considerations discussed in depth in Chapter 2. In particular, their proven ability to provide a successful framework for analysis of many query languages.

**Query Networks and Datalog.** A Datalog query is a generalization of a CQ [92], [2]. See Chapter 2 for description of Datalog.

In the query network model, we look at the large collection of CQs associated with the network...
nodes as a Datalog program. We employ a least-fixpoint bottom-up semantics [92] for the Datalog program composed of all the CQs in the network (Section 3.2 provides formal definitions).

Recently, many Datalog-based languages have been developed for a variety of systems in the fields of networking and distributed systems, computer games, machine learning and robotics, compilers, security protocols and information extraction, as reported in the Claremont report on Database Research [6]. According to this report, the use of declarative languages in these applications has been successful in providing an order-of-magnitude reduction in the size of code, in comparison with other solutions. The report also regards declarative languages as an important step in a suggested prospective development of data management, from a storage service to a programming paradigm.

We believe that this development, combined with the enormous amounts of data and data types currently available to Web 2.0 users (and in other scenarios as well) will inevitably result in a growing need to process large queries. This greatly differs from the prevailing assumption in query processing - that queries are small and data are large. Traditional compilers, optimizers and evaluation techniques are not designed to handle a large query whose size is in the order of magnitude of the data being queried. In particular, rewriting techniques which may increase the query size exponentially (e.g., the Magic Sets optimization for Datalog programs [13]) or add many rules to the program (e.g., the Counting Method [71]) are not applicable in this case.

### 3.1.3 Chapter Outline

Section 3.2 formally defines the Query Network Model. Section 3.3 presents evaluation algorithms for query networks. Section 3.4 briefly discusses possible extensions to the model. Section 3.5 shows the results of experiments conducted with both real and synthetic datasets. The experiments demonstrate the high usefulness of the algorithms. Section 3.6 adds acceptance queries to the model, Section 3.7 presents the corresponding evaluation algorithms, and Section 3.8 shows results of experiments. Section 3.9 analyzes the complexity of several related theoretical problems and discusses variants of the model with a Dunbar number. Section 3.10 surveys related work.
3.2 Preliminaries

We shall use the following definitions:

**A Query Network.** A Query Network is a directed graph \((N, F^0)\) where \(N\) (for Nodes) is a set of participants. \(F^0\) (for Friends) is a set of directed edges between pairs of distinct elements of \(N\). \(F^0 \subseteq (N \times N) \setminus \{(n_0, n_0)\text{ s.t. } n_0 \in N\}\). Every node \(n \in N\) has a query associated with it. The query defines edges of the form \((n, \cdot)\), which are edges such that \(n\) would like to add to the initial set of edges, \(F^0\) (this query may also be the empty query, retrieving no nodes).

**A Query.** A query associated with a network node \(n\), \(q(n)\), is a Datalog rule of the following form. The rule’s head is \(F^+(n, X)\), where \(X\) is a variable. \(F^+\) is an IDB relation which will contain the additions to \(F^0\), which is an EDB relation. We define another relation, \(F\), as \(F = F^+ \cup F^0\). The body of a rule is composed of predicates corresponding to the relation \(F\) and the inequality relation. We require that one of the predicates be of the form \(F(n, Y)\), and that \(X\), the variable in the rule head, appear in one of the \(F\) body predicates. The latter requirement is added for safety (see [92]). Further requirements follow the example.

**Example.** The following query adds to \(F\) tuples of the form \((n, X)\) where \(X\) is a friend of two distinct friends of \(n\) (a constant):

\[
F^+ (n, X) \leftarrow F (n, Y), F (Y, X), F (n, Z), F (Z, X), Y \neq Z,
\]

\[
x \neq n, y \neq n, z \neq n.
\]

We, however, henceforth assume that unless otherwise specified, each two distinct variables imply an inequality predicate between them. We further assume that none of the variables is equal to \(n\). Therefore, the query is abbreviated to:

\[
F^+ (n, X) \leftarrow F (n, Y), F (Y, X), F (n, Z), F (Z, X).
\]

For a query \(q(n)\) we also require that \(n\) is the only constant in the query (that is, all the other arguments are variables).

**Query Graph.** Let the Query Graph of a query \(q(n)\) be the graph whose nodes are the variables and \(n\), the single constant in the query \(q(n)\), and in which a directed edge exists between two variables \(X\) and \(Y\) (respectively, a constant \(n\) and an argument \(Z\)) if a predicate \(F(X, Y)\) (respec-
Figure 3.3: The query graph corresponding to the query $F(n, X) \leftarrow F(n, Y), F(Y, X), F(n, Z), F(Z, X)$.

tively, $F(n, Z)$) occurs in the query body. In the graphical representation of the graph, the variable in the head of the rule appears in a double circle.

**Example.** The query graph corresponding to the query in the previous example is shown in Figure 3.3.

Consider the predicate of the form $F(n, Y)$ which we require in every query body. This predicate contributes to the query graph a node which represents $n$. We require that every query in the query network be such that the nodes in its query graph are all reachable from the node representing $n$.

**Radius.** The *Radius* of a query $q$ is the number of edges in the longest path, that never traverses a node more than once, in the query graph corresponding to $q$. Naturally, we assume that the radius of any query is very small relative to the size of the network.

**Backward Radius (bradius).** Intuitively, the *Backward Radius*, or *bradius*, of a node $n$ in the query network is the maximal distance from another node $m$ such that the query $q(m)$ can ’sense’ the edges whose source is $n$. Formally, we define, $B(n, k) = \{m \in N \mid \text{there exists a path of length } k \text{ from } m \text{ to } n\}$, and $L(n, k) = \{b \mid b \in B(n, k) \text{ and the radius of } q(b) \text{ is at least } k\}$. The *bradius* of node $n$ is the maximal $k$ such that $L(n, k) \neq \emptyset$. Note that the bradius of any node is bounded by the largest radius of a query of the network.

**Example.** Figure 3.4 shows a small example of a query network. Each node has an id (a num-
and is associated with one query, either $q_a$, $q_b$ or $q_c$, whose query graphs also appear in Figure 3.4. We assume that the head of the query is always $F(n, X)$. The radius of the query of Node 2 is 3. The radius of the query of Node 4 is 2. The bradius of Node 8 is 3. The bradius of Node 2 is 1 (additions of the form $(2, \cdot)$ can only be ’sensed’ by nodes 2 and 1).

**Single Evaluation of a Node.** Consider Node 4 in the network illustrated in Figure 3.4. Evaluating once the query associated with this node, $q_b$, inserts the edge $(4, 8)$ to $F$. We call the process of evaluating a node’s query on a given network and subsequently adding a (possibly empty) set of nodes to $F$, an evaluation of a node.

**Exhaustive Evaluation of a Node.** Consider Node 2 in the network after the edge $(4, 8)$ has been added to the original network. After a single evaluation of this node, $(2, 6)$ is in $F$. Once $(2, 6)$ is in $F$, another single evaluation of node 2 will result in the addition of $(2, 8)$ to $F$. Note that this is different than a single evaluation which adds two edges to $F$, since the addition of $(2, 8)$ is done based on the edge $(2, 6)$ that is added in the first evaluation.

The process of evaluating a single query, and this query only, repeatedly and until no edges can be added to $F$, is called an exhaustive evaluation of a node. The network after evaluation is presented in Figure 3.5. The new edges are dashed.
**Round of Network Evaluation.** A *round of network evaluation* is the process of considering all the nodes of a query network in a certain order, and evaluating each node once, in that order. The significance of the order is that edges added previous to, say, the $i$-th single node evaluation, are already in $F$ when the $i$-th evaluation is performed. If node evaluations are exhaustive, then the round is an *exhaustive round* of network evaluation.

**A Fully Evaluated Network.** A network such that a round of evaluation applied to it will not add any edge to $F$ is called *fully evaluated*. We say that a network is a *minimal fully evaluated* network of the original network if it (1) has the same nodes as the original network; (2) contains all the edges in $F^0$; (3) is fully evaluated; (4) is such that there is no proper subset of its edges (and implied nodes) that satisfies (1)-(3). By basic Datalog properties, for a given query network $(N, F^0)$, there exists exactly one minimal fully evaluated network.

**Example.** The network in Figure 3.5 is a fully evaluated network.

Note that the network of Figure 3.5 is a DAG-network (i.e., a network whose graph is a directed acyclic graph). We revisit this example when we discuss a property of DAG-networks in Section 3.3.1.

**The Problem.** The problem for which we propose algorithms is the following. Given a query network $(N, F^0)$ in the input, construct the resulting fully evaluated query network $(N, F)$.
3.3 Network Evaluation Algorithms

We propose three algorithms for evaluating query networks. The Basic evaluation algorithm is a simple evaluation algorithm which performs rounds of network evaluation until the network is fully evaluated. The Backward-Radius Triggering (BRT) evaluation algorithm significantly reduces the number of node evaluations. It does not perform network evaluation rounds. Instead, it identifies nodes whose evaluation is necessary for constructing the fully evaluated network. That is, BRT typically avoids many node evaluations that do not contribute new edges to the network. The Divide and Conquer (DAC) evaluation algorithm partitions the network, evaluates each partition separately and then merges the results. The partitions’ evaluation is completely independent, which makes DAC an algorithm which can greatly benefit from parallelism.

3.3.1 The Basic Algorithm and Related Results

The first algorithm we present is the Basic evaluation Algorithm. This algorithm is a simple algorithm that we use as a baseline for comparison. Pseudo-code for this algorithm is presented as Algorithm 9. Next, we discuss the algorithm and prove two propositions related to the evaluation of query networks in general.

In each iteration, the algorithm performs a single evaluation for each of the nodes in the network. If such a round of evaluation is completed without adding any edge to $F$, the algorithm stops.

**Example.** Let us return to the query network presented in Figure 3.1. Suppose that this network is evaluated by the Basic evaluation algorithm. In the first round of evaluation, evaluating the query associated with Lisa yields a result relation with one tuple, $(Lisa, Marge)$. This tuple is added to $F$, and the stopFlag is toggled to indicate that the main while loop of the algorithm (lines 2 through 11) has to continue for at least another iteration. In this round, the evaluation of the rest of the nodes results in no added edges. In the second round (respectively, third round), the evaluation of Bart’s node (respectively, Lisa’s node) resulted in the addition of the edge $(Bart, Maggie)$ (respectively, $(Lisa, Maggie)$), while the evaluation of the rest of the nodes yields no new edges. In the fourth round, none of the node evaluations results in added edges, and the algorithm stops.

**The Basic Evaluation Algorithm with Exhaustive Rounds.** Note that instead of evaluating
Input: \((N, F^0)\), a query network.

Output: \((N, F)\), a fully evaluated query network.

Method:
1: \(\text{stopFlag} \leftarrow \text{false}\);
2: while \(\text{stopFlag} = \text{false}\)
3: \(\text{stopFlag} \leftarrow \text{true}\);
4: for each \(n\) in \(N\)
5: evaluate \(q(n)\) once, let \(Q\) be the (binary relation) result;
6: if \(Q \setminus F \neq \emptyset\)
7: \(\text{stopFlag} \leftarrow \text{false}\);
8: \(F \leftarrow F \cup Q\);
9: end if
10: end for each
11: end while

Algorithm 9: Basic Network Evaluation Algorithm

each node once, as stated in Line 5 of Algorithm 9, we can instead exhaustively evaluate each node. This may result in reducing the number of rounds necessary for fully evaluating a network, as discussed in Section 3.3.1.2.

3.3.1.1 The Preservation of Cycles Property

Proposition 3.1 Consider a query network \((N, F^0)\). Let \((N, F)\) be the fully evaluated network. Then, if \((N, F)\) contains a cycle, \((N, F^0)\) also contains a cycle.

Proof: Let \(c\) be a set of edges which form a cycle in \((N, F)\). If all the edges in \(c\) are edges in \((N, F^0)\), then \(c\) is a cycle in \((N, F^0)\) and the proposition is proved. If not, then there is at least one edge in \(c\) which is in \((N, F)\) but not in \((N, F^0)\). Let us call such an edge a derived edge. Among the derived edges in \(c\), consider the last edge that the Basic evaluation algorithm applied to \(N\) would add, say \((u, v)\). \((u, v)\) was added as a result of evaluating \(q(u)\). Therefore, there is a (directed) path \(p\), from \(u\) to \(v\), that does not contain \((u, v)\). We delete \((u, v)\) from \(c\) and add the edges of \(p\). \(c\) remains a cycle.
We repeat this process until there are no derived edges in $c$. Every repetition deletes a derived edge from $c$ and replaces it either with non-derived edges or with derived edges that were added to $F$ before the deleted edge. There is a finite number of such derived edges to replace. Eventually, only non-derived edges remain in the cycle $c$.

### 3.3.1.2 One-Round Evaluation of DAG Networks

Consider the network used in the example in Section 3.2, i.e., the network in Figure 3.4, and assume that it is evaluated using the Basic evaluation algorithm with exhaustive rounds. Note that the network graph is a DAG. If the nodes are evaluated in the order of their ids, then in the first round, the edges $(2,6)$ and $(4,8)$ are added. In the second round, $(2,8)$ is added and the third and last round do not add any edge. However, if the order of evaluation is the reverse of the order of ids, then in the first round, $(4,8)$ is added first, and the exhaustive evaluation of node 2 yields $(2,6)$ and $(2,8)$ (in this order). The second and last round does not add any edge.

Next, we show that a network whose graph is a DAG can be fully evaluated in a single exhaustive round of evaluation. Note that knowing that the network is fully evaluated makes a second round redundant.

**Proposition 3.2** Let $(N, F^0)$ be a query network whose graph is a DAG. $(N, F^0)$ can be fully evaluated in one round of exhaustive evaluation.

**Proof:** First, we observe that leaves (i.e., nodes with no outgoing edges) are evaluated, since no query associated with a leaf can be satisfied. Now, as long as there are nodes that are not fully evaluated, we pick one whose descendants are all evaluated, and exhaustively evaluate it. Let the height of a DAG be the maximal number of edges from one of the DAG’s roots to a leaf. We prove by induction on $k$, the DAG’s height, that this process constructs a fully evaluated network.

**Induction Basis.** $k=0$. In this case, there are only leaves in the DAG and the network is therefore fully evaluated.

**Induction Hypothesis.** We assume that for DAGs whose height is up to $k-1$, the process constructs a fully evaluated network.

**Induction Step.** Consider a DAG of height $k$. Consider a node, say $p$, which is $k$ edges distanced from a leaf. We run the process for the DAG rooted at each of $p$’s children. According to the
assumption, each of these (possibly intersecting) DAG-networks is fully evaluated. According to the process, we now exhaustively evaluate $p$. Assume to the contrary that after this exhaustive evaluation, the DAG-network rooted at $p$ is not fully evaluated. Then, there is an edge, which exists in a fully evaluated network but was not added by our procedure. This edge must be of the form $(p,q)$, since the networks rooted at each of $p$’s children are fully evaluated, and the addition of $(p,q)$ can only affect nodes from which $p$ is reachable, which do not exist in the subtrees rooted by the children. However, the exhaustive evaluation of $p$ necessarily adds all the edges of the form $(p,q)$ to the network: such an edge is added based on either (1) original edges of the form $(p,q)$ and edges in the networks rooted at each of $p$’s children or (2) edges the form $(p,q)$ that are added by $p$’s query and edges in the networks rooted at each of $p$’s children. The contradiction implies that the DAG-network rooted at $p$ is fully evaluated.

Note that the DAG structure of the graph ensures that if there is an unevaluated node $p$ in the network, then there is also an unevaluated node whose descendants are all evaluated (for example, one of $p$’s descendants). This is why the procedure could be defined without referring to the DAG structure.

3.3.2 The Backward-Radius Triggering Algorithm

Consider, again, the evaluation of the network illustrated in Figure 3.1 using the Basic algorithm. Four rounds of evaluation resulted in 28 single node evaluations, whereas only three of these evaluations actually yield edge addition. The Backward-Radius Triggering (BRT) evaluation algorithm will reach a fully evaluated network by usually performing a significantly lower number of single node evaluations.

BRT takes $k$, the maximal (query) radius in the network, as input. $k$ is used as a bound on the backward radius of the nodes. In BRT, when an edge, say $(u,v)$, is added, only nodes whose queries can ‘sense’ the addition are considered for another evaluation. These nodes are such that there exists a (directed) path in the query network, whose length is less than the backward radius, between them and $u$. The backward radius is bound by $k$.

Note that the important model feature is that $k$ is small relative to the network. Merely knowing what $k$ is could be ascertained in the first round of BRT. Therefore, passing $k$ as a parameter is not
essential to the algorithm. For simplicity, we give $k$ as input to BRT.

Pseudo-code for the BRT evaluation algorithm appears as Algorithm 10. First, all the nodes are put in the set $R$, and a single node evaluation is performed for every node $n$ in $R$. This is in fact a round of network evaluation. For every node $n$ whose evaluation results in the addition of an edge (or multiple edges) to $F$, the set $\{m \in B(n, l) | l < k\}$ is computed (see definition in Section 3.2) and added to $P$. $P$ replaces $R$ and the evaluation continues until $R$ is empty.

```plaintext
Input: $(N, F^0)$, a query network; $k$, maximal query radius.
Output: $(N, F)$, a fully evaluated query network.
Method:
1: $R \leftarrow N$
2: while $R \neq \emptyset$
   2: $P \leftarrow \emptyset$
   3: for each $n \in R$
      3: evaluate $q(n)$, let $Q$ be the (binary relation) result;
      6: if $Q \setminus F \neq \emptyset$
      7: $P \leftarrow P \cup \{m \in B(n, l) | l < k\}$
      8: $F \leftarrow F \cup Q$
      9: end if
   10: end for each
11: $R \leftarrow P$
12: end while

Algorithm 10: Backward Radius Triggering Algorithm
```

**Example.** Consider again the query network presented in Figure 3.1. In this network, $k = 3$. If evaluated with the BRT algorithm, the first iteration will consider all the nodes for a single evaluation, and the edge $(Lisa, Marge)$ will be added. $B(Lisa, 1) = \{Bart\}$ and $B(Lisa, 2) = \emptyset$. Therefore, $P = \{Bart\}$. The evaluation of the single node in $P$ results in the addition of the edge $(Bart, Maggie)$, and at the end of this iteration, $P = \{Lisa\}$. In the next iteration, the edge $(Lisa, Maggie)$ is added, and $P = \{Bart\}$. In the next iteration, no edge is added. As a result, $P = R = \emptyset$ and the algorithm stops.
Comparing Basic and Backward-Radius Triggering. The total number of single-node evaluations in the latter example is (broken by iteration) $7 + 1 + 1 + 1 = 10$. As shown above, the Basic algorithm performs, on the same network, 28 evaluations. The benefit of saving single rule evaluations comes at the price of computing the sets $B(n, k)$. Also, Basic does not use $k$.

3.3.3 The Divide and Conquer Algorithm (DAC)

We would like to be able to process large networks. In the DAC evaluation algorithm, we take advantage of the clustered nature of social networks in order to partition the network into networks of manageable sizes. Generally speaking, social networks have a structure in which participants have more links to participants within their community than to individuals from other communities [58] (see more in Section 3.10). We use existing knowledge and algorithms for graph partitioning in order to partition the graphs to parts, with a relatively small number of edges between them. This partitioning enables us to fully process small, dense sub-networks, taking advantage of locality of reference and minimizing work for a merge step.

Partitioning Algorithm. A partitioning algorithm for a query network takes a query network, say $(N, F^0)$, as input and produces a number of query networks as output. $N$ is partitioned into (non-overlapping) sets of nodes. Each such set $N_i$, and the edges in $F^0$ between the nodes in $N_i$ form a query network in the output. Crossing edges are edges in $F^0$ that are in none of the created networks.

Like BRT, DAC takes $k$, the maximal radius of query in the network, as input. Beside $k$, DAC takes as input a graph partitioning algorithm ($A_1$), the number of partitions ($p$), a network evaluation algorithm ($A_2$), and a partition-matching algorithm, $A_3$, which takes a set of subnetworks as its input and outputs a set of sets of subnetworks.

Pseudo code for DAC appears as Algorithm 11. DAC operates as follows. First, the partition algorithm partitions the network into smaller networks. Every part is evaluated separately using the given network evaluation algorithm $A_2$ (Line 2). For example, in our implementation, $A_2$ is BRT. DAC could also call itself as $A_2$ (and stop the recursion according to a threshold). Then, a
match-making procedure \( A_3 \), which matches pairs of partitions for merging (or sets of more than two members), is invoked (Line 4). An example for such a procedure is \textit{matchPairs}, which matches two networks such that the number of crossing edges between them is maximal. This is a heuristics to choose which, and how many, partitions to merge; other heuristics may also be used here. Then, the rest of the networks are considered, and another pair is matched and so on. The match making continues until less than two networks remain unmatched, i.e., one.

Each pair is merged into one network (Line 7). The nodes of the new network are the union of the nodes of the networks being merged. The edges are the union of the edges of the networks being merged, as well as the crossing edges between the merged networks. Due to the addition of crossing edges, the merged network is not necessarily fully evaluated. The \textit{merge&eval} procedure evaluates the merged network, first by evaluating all the nodes \( n \) that are sources of the cross edges and the nodes within \( B(n, l), l < k \) for each such node \( n \), in order to include all the nodes in their backward radius. Like in BRT, any such node whose evaluation results in the addition of new edges triggers the evaluation of the nodes potentially in their backward radius and so on until a fixpoint is reached.

\textbf{Example.} Consider the network sketches in Figure 3.6. Dark grey (respectively, light grey) represents a fully evaluated (respectively, non fully evaluated) network. In \( (a) \), a network partitioned into four parts is presented. Only crossing edges are presented. In \( (b) \), each of the four parts is fully evaluated, ignoring crossing edges. In \( (c) \), the matchmaking result is illustrated. The parts with maximal number of crossing edges between them were paired. Note that the pairs of networks are not yet evaluated. In \( (d) \), the merged and fully evaluated pairs are shown. Another matchmaking step is \( (e) \), and the fully evaluated network is \( (f) \).

Note that as in BRT, \( k \) need not be known in advance. Here also, \( k \) can be ascertained for each initial partition (i.e., before any merge has been done) on the first round and maintained for further evaluations as the maximum value of each merged pair (note however that \( k \) for different parts may be lower than the global \( k \)). For simplicity, we give \( k \) as input to DAC.
**Input:** \((N, F^0)\), a query network;

\(k\), maximal query radius;

\(A_1\), a graph-partitioning algorithm;

\(p\), number of partitions;

\(A_2\), a network evaluation algorithm.

\(A_3\), a partition-matching algorithm.

**Output:** \((N, F)\), a fully evaluated query network.

**Method:**

1: invoke \(A_1\) to partition \((N, F^0)\) into \(p\) subnetworks (s.n.),

2: let \(R\) be the set of s.n., \{\((N_i, F_i^0)\) s.t. \(1 \leq i \leq p\)\};

3: invoke \(A_2\) evaluation on each \(r \in R\);

4: while \(\exists (u, v) \in F^0\) s.t. \(u\) and \(v\) are in different s.n.

5: \(P \leftarrow A_3(R); //\text{see for example matchPairs}\)

6: for each \((r_i, ..., r_j) \in P //\text{matched merge}\)

7: \(R \leftarrow R \setminus \{r_i, ..., r_j\};\)

8: \(R \leftarrow R \cup \text{merge&eval}(r_i, ..., r_j); //\text{see procedure below}\)

9: end while

Algorithm 11: Divide and Conquer for Networks with Acceptance Queries

### 3.4 Extensions

The model presented so far is the minimal model needed to exhibit our concept of large queries in a Web 2.0 environment. However, in order to be used in real systems, some extensions to the model should be considered. We point out directions which received positive feedback from referees of related publications. Development and Implementation are left for future work.

**Data integration.** The web environment is diverse in data types. Of particular interest is the interplay between structured and unstructured data [6]. A natural extension to our model will be to also process XML data. Every node in the model, in addition to having a list of friends, will also have an XML document associated with it. Inspired by [79] and [78], a query that refers to
**procedure** matchPairs($R$) $// R$ is a set of networks

1: for each crossing edge $(u, v)$ w.r.t. the networks in $R$
2: let $r_u$ (respectively, $r_v$) be $u$’s (respectively, $v$’s)
   - network;
3: initialize a counter $c_{u,v}$ to 0, if not exists;
4: $c_{u,v} \leftarrow c_{u,v} + 1$;
5: end for each
6: $P \leftarrow \emptyset$;
7: while there is more than one network in $R$
8: add to $P$ a pair $(r_v, r_u)$ s.t. $c_{u,v} + c_{v,u}$ is maximal;
9: delete $r_v$ and $r_u$ from $R$;
10: end while
11: Add unmatched singleton, if exists, to $P$; return $P$;

**Algorithm 12:** Match Pairs Procedure - An example for $A_3$

---

Figure 3.6: Divide and Conquer Example.
procedure merge&eval((N_1, F_1),..., (N_n, F_n))
1: N ← N_1 ∪ ... ∪ N_n;
2: F ← F_1 ∪ ... ∪ F_n;
3: let C be the set of crossing edges between
   - the networks;
4: F ← F ∪ C; Q ← ∅;
5: M ← \{n_1|(n_1, n_2) ∈ C\}; // sources of new edges
6: G ← \{g|g ∈ B(m, l), 1 < l < k, m ∈ M\}; // nodes possibly affected by the new edges
7: for each m ∈ M
8: evaluate q(m), let Q be the (binary relation) result;
9: F ← F ∪ Q;
10: end for each
11: while G ≠ ∅
12: P ← ∅;
13: for each g ∈ G
14: evaluate q(g), let Q be the (binary relation) result;
15: if Q \ F ≠ ∅
16: P ← P ∪ \{m ∈ B(g, l)|l < k\}
17: F ← F ∪ Q;
18: end if
19: end for each
20: G ← P;
20: end while
21: return (N, F);

Algorithm 13: Merge and Eval Procedure.
the XML document and to the structured data will have the following form:

\[ F^+(n, X) \leftarrow F(n, Y), F(Y, X), F(n, Z), F(Z, X), \]

\[ \text{xpath}(X, '/data/language[text()="French"]') \]

The \text{xpath} predicate will be satisfied by nodes whose associated XML document satisfies the expression 

'\text{/data/language[text()="French"]}'.

As in SQL/XML [88], the expression is evaluated at the document root node. Query languages other than XPath can be used to query XML data, and other data types with their query languages may be used in the CQ model.

**More than one query.** A realistic policy for the additions of connections to friends’ lists in a social network will most likely require more than one query. A natural extension of our model would be to have a set of queries, or even a Datalog program (or programs), associated with each node. This addition raises interesting evaluation problems. For example, how to order the evaluation of queries associated with the same node so as to minimize the number of evaluation rounds. Also, the radius and backward radius cannot be used as they are currently defined.

**Not using a global radius.** Currently, the radius is a parameter of an evaluated network or part of it. As explained above, the radius is either gathered from the network or passed as input. However, we use the radius as a bound on the backward radius of network nodes. A more sophisticated algorithm will efficiently initialize and maintain, for every node, its backward radius, and avoid evaluating nodes that are within the (globally) highest backward radius in the network when it is unnecessary for the particular node given its backward radius.

### 3.5 Implementation and Experimentation

We implemented the presented algorithms in a system prototype. The prototype was programmed in Java, using the open-source DBMS MySQL, version 5.0. The experiments were carried out on a machine with a Pentium CPU of 1.5GHZ and 1GB of RAM, running the Windows XP operating system. We experimented with the algorithms, both on synthetic query networks as well as on query networks derived from the DBLP data, which reflect social behavior.
3.5.1 Synthetic Datasets

Our synthetic datasets are built as follows. Consider the illustration in Figure 3.7. The figure shows a number of clusters. Connections may exist between participants in the same cluster or between participants in neighboring clusters. The probability that participants in the same cluster are connected is higher than the probability that two participants in neighboring clusters are connected. The clusters form a cycle in which every city has exactly two neighbors, one to the right and one to the left.

The graphs are synthesized as follows. The number of clusters and the number of nodes in each cluster are given as input. For every node \( n_1 \), we pick at random a node \( n_2 \) s.t. \( n_1 \neq n_2 \) from the same cluster, and add an edge \((n_1, n_2)\). Then, every node has probability \( \alpha \) to be connected to each of the other nodes in the cluster. At least one node, plus up to additional \( \beta \) nodes per hundred in a cluster, are connected each to a randomly chosen participant in the immediate neighboring cluster to the left, and a similar number is connected to participants in the immediate neighboring cluster to the right.

The queries used in our experiments are \( q_a \) and \( q_b \) from the example in Section 3.2, as well as an unsatisfiable query. Partitioning for the DAC algorithm is done using the Metis software package [66]. We also show results for partitioning the graph to parts such that each part corresponds to a cluster.

**Experiment 1.** In this experiment, we create 10 datasets. The \( i \)-th dataset has \( 5 \times i \) clusters. Each cluster has 180 nodes. \( \alpha = 1/200, \beta = 1 \). Nodes are randomly associated with either \( q_a \) or \( q_b \) (with probability 0.5 for each of the queries).
Consider Figure 3.8. The running time results for each of the ten datasets appear as four figures (from left to right): DAC time using the Metis partitioner (DAC-Metis), DAC time using the initial, predefined partition to clusters (DAC-Pre), BRT time, Basic time. The datasets are ordered according the IDB/EDB ratio, which is the number of added edges divided by the number of original EDB edges. In the appendix, we list more details about these datasets. The datasets numbers correspond to their EDB size.

Basic evaluation is the slowest, except for the smallest dataset. As for the remaining two algorithms, the results clearly show that DAC outperforms BRT when the number of added edges is high relative to the number of EDB edges. BRT is better when relatively few edges are added in reaching a fully evaluated network. Partitioning using Metis or partitioning according to the initial clusters does not make a significant difference for DAC’s performance (partitioning times are included in the total DAC time). Figure 3.9 orders the same results by the number of added edges (namely, the IDB size). Figure 3.10 orders the results by the size of the EDB. The results show a clear correlation between the IDB/EDB size ratio and the benefit of using DAC rather than BRT. A similar correlation exists between the IDB size and the benefit of using DAC.

**Experiment 2.** In this experiment, we create 10 datasets. Again, the $i$-th dataset has $5 \times i$ clusters. Each cluster has 150 nodes. $\alpha = 5/900$, $\beta = 2$ and $q_a$ and $q_b$ are independently and evenly distributed among the nodes. Figure 3.11 shows the time results for the three algorithms (partitioning for DAC is done using Metis). The results are ordered by the IDB/EDB ratio, which also appears in a table for each dataset. Basic is the slowest algorithm, and DAC is typically faster.
Figure 3.9: Time results of Experiment 1. Datasets ordered by IDB size.

Figure 3.10: Results of Experiment 1. Datasets ordered by EDB size.

than BRT, in particular for large datasets.

**Experiment 3.** In this experiment, each cluster has 225 nodes, $\alpha = 1/200$, $\beta = 2$ and $q_a$, $q_b$ and an unsatisfiable query are evenly distributed among the nodes. Figure 3.12 shows the time results of the three algorithms (partitioning for DAC is done using Metis). Here too, Basic is the slowest, and DAC outperforms BRT, in particular for large datasets.

### 3.5.2 DBLP Datasets

We also conducted experiments on data derived from the DBLP datasets [28]. We use DBLP in order to capture patterns of social behavior that are reflected in the collaborations between authors. We extracted 260360 papers with their 260801 authors. For each pair of authors we also keep the
number of papers that they published together.

We construct the datasets for the experiments as follows. We start with one author, and perform a crawling-like procedure. We retrieve five of the author’s collaborators (if there are less than five, all are retrieved). Then, we retrieve five collaborators of each of the five retrieved in the first step, and so on until the desired number of authors is reached.

The input network is constructed as follows. Each author is represented by a node. If the share of publications in common to \( u \) and \( v \) is at least \( \alpha \) of \( v \)’s total number of publications then there is an edge from \( u \) to \( v \). That is, if \( u \) participated in a high enough share of \( v \)’s publications (\( \alpha \) percent or more), the edge \((u, v)\) is added.

**Experiment 4.** In this experiment, the graph consists of 1300 nodes, partitioned into 16 parts.
(using Metis). We create the datasets by gradually lowering $\alpha$ from 33\% to 14\%, in order to increase the number of edges in the EDB. Figure 3.13 shows the running time results of the three algorithms on datasets generated as described above. The results are in line with the results of the synthetic datasets.

**Experiment 5.** In this experiment, the graph consists of 1400 nodes, partitioned into 16 parts (using Metis). Again, we create the datasets by gradually lowering $\alpha$ from 33\% to 14\%. Figure 3.14 shows the result of this experiment, which are in line with the results of Experiments 1 through 4. Results for the Basic algorithm were not included in the figure and are instead given below, so as not to overload the figure with high numeric values. The results are again in line with the results of the previous experiments.

Results for the Basic algorithm in Experiment 5 are: Dataset 1: 6.5 sec, Dataset 2: 30.4 sec,
Dataset 3: 36.8 sec, Dataset 4: 56.0 sec, Dataset 5: 190.1 sec, Dataset 6: 2771.5 sec.

3.6 Query Networks with Proposal and Acceptance Queries

In this section, we add acceptance queries to the Query Network model, which captures the elementary social network interaction of proposing/accepting a friendship connection to/from another participant. Participants have proposal and acceptance rules, defined as queries. As in the basic Query Network model, these rules are defined once for long term operation. Proposal and acceptance rules react to the developments in the network by interacting with each other and adding connections. By this, they in fact serve as a simple protocol, opening the door to more complex protocols and automation tools for social networks, such as the tools discussed in Chapter 4 of this thesis.

While the most typical connection in a social network is the friendship connection, the evolution of social networks exposes many new types of connections: between users and event, fans and stars, other business-oriented connections, tags and photos, and more. In particular, connections relevancy may well be temporary. For example, connection to an event in the past may stop being relevant for most everyday uses. This further emphasizes the importance of protocol-like automation for the creation of any type of connections, for which automating the creation of friendship connections is a primary example.

The Query Network with Proposal and Acceptance Queries, which is a generalization of the query network model, captures this query-based protocol-like setup. A new edge has to be proposed by a proposal query and accepted by an acceptance query. Unlike the Basic Query Network model, no single query has an immediate effect on the graph. The model is graph-based, which makes it useful for connections beyond the friendship connections, as discussed above. In addition, nodes have attributes, which enrich the possibilities to formulate queries.

In a real-world system, queries may have other semantics than automatic creation of connections. For example, an acceptance rule can be a filtering rule, leaving the final decision to the user.
3.6.1 Example

Consider the network of eight tennis players in Figure 3.15. Connections are represented by directed edges (an issue we further discuss in Section 3.1.2). Original edges are solid. Edges to be added by network evaluation are dashed. The table in Figure 3.15 contains basic information on the players.

Proposals. Some proposal rules in the network follow.

- Bob (player 1) would like to connect to every player \( p \) such that two distinct friends of Bob are connected to \( p \).
- Alice (player 2), would like to connect to players whose rank is at most 7, and to which there exists a 3-edge path from her (i.e., they are a friend of a friend of a friend).
- Charlie (player 3) would like to connect to every player connected to him through two distinct paths, such that one path is of length 2 edges and the other - of length 3.

Acceptances. Some acceptance rules follow.

- Irene (player 6) accepts proposals from any player \( p \) that has a two-edge path to her, say \( p \to q \to Irene \), such that \( q \) is bidirectionally connected to a player whose rank is 1.
- Diana (player 4) accepts a connection only from men with a score exceeding 80.
- Ken (player 8) is willing to accept a connection from any player whose score exceeds 80.
- Jacob (player 7) accepts connections from everybody.
- For simplicity, let us assume that the proposals and acceptances of the rest of the participants define empty sets.

Addition of Connections.

- Bob would like to connect to Irene (player 6). Bob satisfies Irene’s first condition, since there exists a two-edge path from Bob to Irene (through 3 or 5). However, neither player 3 nor 5 has a bidirectional connection to a player of rank 1. Therefore, no edge is added.
- There exists a path of length 3 from Alice to Diana, Irene and Ken (players 4,6,8). Alice proposes a connection to 4 and 8, but not 6 since its rank is more than 7. Alice is accepted only by 8, as she is a female. An edge \((2,8)\) is therefore added.
- Based on the edge \((2,8)\), Charlie (player 3) proposes a connection to Ken and to Helen. Ken accepts the proposal, since Charlie’s score is more than 80. Helen does not, since her acceptance query defines the empty set. \((3,8)\) is added.
Figure 3.15: The Tennis Players Network and players’ data. Solid edges are original edges, and dashed edges are edges added by evaluation.

- Since the edge (3,8) exists, Charlie has a bidirectional connection to a player of rank 1 (Ken). Bob is therefore accepted by Irene, and the edge (1,6) is added.
- As a consequence, Alice proposes a connection to Jacob. It is accepted, and the edge (2,7) is added.

The addition of (2,8) led to the addition of (3,8), which led to the addition of (1,6). (1,6) led to the addition of (2,7). When we first considered player 2, (2,7) could not be added. Added edges appear as dashed edges in Figure 3.15.

### 3.6.2 Formalization

As the basic Query Network, a **Query Network with Acceptance Queries** is a directed graph $(N, F^0)$ where $N$ (for Nodes) is a set of participants. $F^0$ (for Friends) is a set of directed edges between pairs of distinct elements of $N$, $F^0 \subseteq (N \times N) \setminus \{(n_0, n_0) \text{ s.t. } n_0 \in N\}$. Every node $n \in N$ has two queries associated with it, one proposal query and one acceptance query. Nodes also have attributes which contain information related to the node. The limit of one proposal and one acceptance query is only for the sake of discussion simplicity. In general, multiple queries can be used.

The proposal query defines edges of the form $(n, \cdot)$, which are edges such that user $n$ would like to propose as candidates for addition to the graph. The acceptance query defines edges of the form $(\cdot, n)$, which are edges such that $n$ would like to accept, if proposed as candidates by other nodes. An edge $(u,v)$ proposed by $u$ and accepted by $v$ is added to the initial set of edges, $F^0$, as defined below.
Proposal and Acceptance Queries. A proposal query associated with a network node (user) \( n, q_p(n) \), is a Datalog rule or rules whose head is \( F_p(n, X) \), where \( X \) is a variable \(^1\). An acceptance query \( q_a(n) \) is a Datalog rule or rules whose head is \( F_a(X, n) \). We define another Datalog rule, called the consensus rule, which is not associated with any of the nodes. The consensus rule is:

\[
F^+ (Z, W) \leftarrow F_p(Z, W), F_a(Z, W)
\]

\( F_a \) and \( F_p \) are IDB relations which contain edges that are candidates to be added to the network. \( F^+ \) is an IDB relation which contains the additions to \( F^0 \), which is an EDB relation. We define another relation, \( F \), as \( F = F^+ \cup F^0 \).

Queries Structure. The body of an acceptance query is composed of predicates corresponding to the relation \( F \) and to nodes’ attributes, and of the inequality predicate (examples follow). In a proposal query at node \( n \), we require that one of the body predicates be of the form \( F(n, Y) \), and that \( X \), the variable in the rule head, appear in one of the \( F \) body predicates. The latter requirement is added to ensure safety (see [92]). Acceptance queries are only required to be safe. Further requirements follow the next example.

Examples. (1) The following is \( n \)'s proposal query, proposing a connection to every participant who is a friend of two distinct friends.

\[
F^+_p(n, X) \leftarrow F(n, Y), F(Y, X), F(n, Z), F(Z, X), Y \neq Z, X \neq n, Y \neq n, Z \neq n.
\]

We henceforth assume that unless otherwise specified, two distinct variables imply an inequality predicate between them.

(2) The following is \( n \)'s acceptance query, expressing the same policy as Irene’s in the example in Section 3.6.1 (namely approving a connection proposed by a node \( X \))

\[
F^+_a(X, n) \leftarrow F(Y, n), F(X, Y), F(Y, Z), F(Z, Y), Z.\text{rank}=1.
\]

For a query \( q_a(n) \) or \( q_p(n) \) we also require that \( n \) is the only constant in the query which represents a node (that is, all the other arguments which represent nodes are variables). Constants may appear in the context of attributes.

Query Graph. Let the Query Graph of a query \( q(n) \) (either proposal or acceptance) be the

\(^1\)The subscript \( p \) stands for propose, and \( a \) - for accept.
graph whose nodes are the variables and \( n \), the single constant in the query \( q(n) \), and in which a directed edge exists between two variables \( X \) and \( Y \) (respectively, a constant \( n \) and an argument \( Z \)) if a predicate \( F(X, Y) \) (respectively, \( F(n, Z) \) or \( F(Z, n) \)) occurs in the query body. In the graphical representation of the graph, the variable in the head of the rule appears in a double circle. For a proposal query, we require that the nodes in its query graph are all reachable, via a directed path, from the node representing \( n \). In addition, only attributes of nodes that appear in the query graph (that is, attributes of nodes that appear as arguments in a predicate corresponding to \( F \)) are referenced.

**Example.** Figure 3.16 illustrates the graphs corresponding to the queries in the previous example.

**Radius.** The *Radius* of a proposal query \( q_p \) is the number of edges in the longest path, that never traverses a node more than once, in the query graph corresponding to \( q_p \). Naturally, we assume that the radius of any query is very small as compared to the size of the network. We address again the relation between the network and the radius later on.

**Rewriting of Acceptance Queries.** Unlike proposal queries, the acceptance query graph need not have a pattern structure with a radius. For example, in Section 3.6.1, Diana accepts all men with a score exceeding 80. Therefore, an acceptance query can be satisfied by a large number of nodes. However, in the context of a network in which the maximal proposal query radius is \( k \), a proposal to node \( v \) comes only from nodes distant at most \( k \) edges from \( v \). In order to avoid large intermediate results in our algorithms, we add to every acceptance query a condition which requires accepted nodes to be distant at most \( k \) edges from the accepting node.

**Single Evaluation of a Node.** We call the process of evaluating the two queries of a given network node and subsequently adding a (possibly empty) set of nodes to \( F_a \) and \( F_p \), a *single evaluation of a node*.

**Round of Network Evaluation.** A *round of network evaluation* is the process of considering all the nodes of a query network in a certain order, and evaluating each node once, in that order.
Then, evaluating the consensus rule.

A Fully Evaluated Network. A network such that a round of evaluation applied to it does not add any edge to $F$ is called fully evaluated. We say that a network is a minimal fully evaluated network of the original network if it (1) has the same nodes as the original network; (2) contains all the edges in $F^0$; (3) is fully evaluated; (4) is such that there is no proper subset of its edges (and implied nodes) that satisfies (1)-(3). Note that for a given query network $(N, F^0)$, there exists exactly one minimal fully evaluated network. This follows from basic Datalog properties [92].

The Problem. As in the absence of acceptance queries, the problem for which we propose algorithms is to construct, given a query network $(N, F^0)$ in the input, the (unique) minimal fully evaluated network $(N, F)$.

3.7 Evaluation Algorithms for Networks with Acceptance Queries

We propose four evaluation algorithms for query networks with acceptance queries. The Basic (for Networks with Acceptance Queries) Algorithm (or, Basic-a) is a naive algorithm which performs rounds of network evaluation (for both proposal and acceptance queries as defined above) until the network reaches a fixpoint. The Propose-Accept Algorithm significantly reduces the number of acceptance query evaluations. The PA with Backward Radius Algorithm does not perform rounds,
but identifies proposal and acceptance queries that need be evaluated. The Divide and Conquer (for Networks with Acceptance Queries) (or, DAC-a) Algorithm takes advantage of the clustered nature of social networks graphs by evaluating parts of the network separately and independently, and merging the results.

3.7.1 Basic-a Evaluation Algorithm

The Basic-a Algorithm is a naive algorithm that serves as baseline for comparison. Pseudo-code for Basic-a is presented as Algorithm 14. In every iteration, Basic-a performs a single node evaluation for each network node. Edges corresponding to accepted proposals are added. The algorithm stops when a round is completed without adding any edge.

```
Input: \((N, F^0)\), a query network.
Output: \((N, F)\), a fully evaluated query network.
Method:
1: stopFlag ← false;
2: while stopFlag = false
3:   stopFlag ← true;
4:   \(Q_p(\cdot, \cdot) \leftarrow \emptyset\);
5:   \(Q_a(\cdot, \cdot) \leftarrow \emptyset\);
6:   for each \(n\) in \(N\)
7:     evaluate \(q_p(n)\) once, add the result to \(Q_p(\cdot, \cdot)\);
8:     evaluate \(q_a(n)\) once, add the result to \(Q_a(\cdot, \cdot)\);
9: end for each
10: \(Q \leftarrow Q_a \cap Q_p\);
11: if \(Q \setminus F \neq \emptyset\)
12:   stopFlag ← false;
13: end if
14: \(F \leftarrow F \cup Q\);
15: end while
```

*Algorithm 14: Basic-a Evaluation Algorithm*

116
Example. Suppose that the network in Figure 3.15 is evaluated by the Basic-a Algorithm, and that queries are evaluated in the order of node ids. In the first round, the edge (2,8) is added. In the second round, (3,8) is added. Note that (3,8) is not added in the first round, since the edge (2,8) (used in the creation of (3,8)) is not added until the end of the first round. In the third (respectively, fourth) round, (1,6) (respectively, (2,7)) is added. There are no additions in the fifth round, and the algorithm therefore stops.

3.7.2 Propose-Accept Evaluation Algorithm

In every round of the Basic-a algorithm, operating on the network in Figure 3.15, 16 queries corresponding to 8 nodes are evaluated. The total number of evaluated queries is 80, of which 40 are acceptance queries. Out of these 40 evaluations of acceptance queries, only 4 resulted in the addition of an edge. The Propose-Accept (PA) Algorithm reduces the number of evaluated acceptance queries. Pseudo-code for PA appears as Algorithm 15. In an evaluation round, all proposal queries are evaluated. Acceptance queries are then evaluated only for the proposed nodes.

Example. Consider again the network in Figure 3.15. In every round of PA, all 8 proposal queries are evaluated. However, in the first round, only 4 acceptance queries are evaluated (of nodes 4, 6, 5 and 8), in the second (respectively, third, fourth) - 4 acceptance queries (respectively, 2, 3) and none in the fifth. The total is 13 evaluated acceptance queries.

3.7.3 Propose-Accept with Backward-Radius Triggering

We now turn to also reduce the number of proposal queries evaluations. Forty proposal queries are evaluated in the previous examples. Four of them actually yield addition of edges. The PA with Backward Radius Triggering algorithm (PABRT) also reduces the number of proposal queries evaluations, reaches a minimal fully evaluated network, and typically evaluates a significantly smaller number of proposal queries than the previous algorithms.

Backward Radius. Intuitively, the Backward Radius (BR) of a network node $n$ is the maximal distance from another node $m$ such that a predicate in the proposal query $q_p(m)$ may be satisfied by edges emanating from $n$. Formally, we define, $B(n,k) = \{m \in N | \text{there exists a path of length } k \text{ from } m \text{ to } n\}$, and $L(n,k) = \{b | b \in B(n,k) \text{ and the radius of } q_p(b) \text{ is at least } k\}$. The BR of
**Input:** $(N, F^0)$, a query network.

**Output:** $(N, F)$, a fully evaluated query network.

**Method:**

1: $\text{stopFlag} \leftarrow \text{false}$;

2: while $\text{stopFlag} = \text{false}$

3: $\text{stopFlag} \leftarrow \text{true}$;

4: $Q_p(\cdot, \cdot) \leftarrow \emptyset$;

5: $Q_a(\cdot, \cdot) \leftarrow \emptyset$;

6: for each $n$ in $N$

7: evaluate $q_p(n)$ once, add the result to $Q_p(\cdot, \cdot)$;

8: end for each

9: for each $m$ s.t. $(\cdot, m)$ in $Q_p$

10: evaluate $q_a(m)$ once, add the result to $Q_a(\cdot, \cdot)$;

11: end for each

12: $Q \leftarrow Q_a \cap Q_p$;

13: if $Q \setminus F \neq \emptyset$

14: $\text{stopFlag} \leftarrow \text{false}$;

15: $F \leftarrow F \cup Q$;

16: end if

17: end while

**Algorithm 15:** Propose-Accept Evaluation Algorithm

Node $n$ is the maximal $k$ such that $L(n, k) \neq \emptyset$. So, if an edge from $n$ to some edge is added, only proposal queries in nodes within distance $\text{BR}$ may ‘sense’ it. The BR of any node is bounded by the maximal radius of a propose query in the network.

PABRT takes $k$, the maximal radius of a proposal query, as input. $k$ is used as a uniform bound on the BR of all the nodes. In PABRT, when an edge $(u, v)$ is added, only nodes whose proposal queries can potentially ‘sense’ the addition are considered for another evaluation. These nodes are such that there exists a (directed) path, whose length is less than the BR, between them and $u$. Note that we use the model feature that $k$ is small relative to the network. Passing $k$ as a parameter is
not essential for the algorithm, since \( k \) can be ascertained in the first round of PABRT, which does not use \( k \). For simplicity, we pass \( k \) explicitly as a parameter.

Pseudo-code for PABRT appears as Algorithm 16. First, all the nodes are put in the set \( R \), and the proposal query of each node is evaluated. As in PA, relevant acceptance queries are evaluated, and new connections are added to \( F \). For every node \( n \) which is a source of such an added edge (or multiple edges), the set \( \{ m \in B(n, l) | l < k \} \) is computed and added to \( P \). \( P \) replaces \( R \). The evaluation stops when \( R \) is empty.

**Example.** Consider, once again, the query network presented in Figure 3.15. In this network, \( k = 3 \). If evaluated with PABRT, after the first pass, the edge \((2,8)\) is added to the network, and \( P = \{1,3,5,8\} \). Based on \( P \), \((3,8)\) is added and \( P = \{1,2,5,6,8\} \). Then, \((1,6)\) is added and \( P = \{2,3,6\} \). Eventually, \((2,7)\) is added, \( P = \emptyset \) and the algorithm stops. The number of proposal queries evaluated in this example is, broken by the different contents of \( P \): 11 + 4 + 5 + 2 = 22 (as compared to 40 in the previous examples).

### 3.7.4 Divide and Conquer for Networks with Acceptance Queries

As discussed above, social networks have a clustered nature, that we would like to take advantage of, in order to evaluate the network more efficiently. Pseudo code for the Divide and Conquer Algorithm for Networks with Acceptance Queries (DAC-a) appears as Algorithm 17. Like DAC, DAC-a takes a number of values as parameters: the maximal radius \( k \), a graph partitioning algorithm \( A_1 \), the number of partitions \( p \), a network evaluation algorithm \( A_2 \), and a partition-matching algorithm, \( A_3 \), which takes a set of subnetworks as the input and outputs a set of sets of subnetworks.

First, \( A_1 \) is invoked and produces \( p \) subnetworks. \( A_2 \) evaluates each of them separately. Then, sets of subnetworks are matched using \( A_3 \). An example for a partition-matching algorithm is the Pair Matching algorithm shown as Algorithm 12, which outputs sets of up to two subnetworks as follows. The two partitions with the largest number of crossing edges between them are matched. Another such pair is found among the rest of the partitions, and so on. In general, \( A_3 \) can produce any subset, not necessarily doubletons (as Algorithm 12 does). Each set of subnetworks is merged
Input: \((N, F^0)\), a query network; \(k\), maximal proposal query radius.

Output: \((N, F)\), a fully evaluated query network;

Method:

1: \(R \leftarrow N\)

2: while \(R \neq \emptyset\)

3: \(P \leftarrow \emptyset\)

4: \(Q_p(\cdot, \cdot) \leftarrow \emptyset\)

5: \(Q_a(\cdot, \cdot) \leftarrow \emptyset\)

6: for each \(n \in R\)

7: evaluate \(q_p(n)\) once, add the result to \(Q_p\);

8: end for each

9: for each \(m\) s.t. \((\cdot, m)\) in \(Q_p\)

10: evaluate \(q_a(m)\) once, add the result to \(Q_a\);

11: end for each

12: \(Q \leftarrow Q_a \cap Q_p\)

13: if \(Q \setminus F \neq \emptyset\)

14: \(P \leftarrow P \cup \{m \in B(f, l)|l < k \land f \in (Q \setminus F)\}\)

15: \(F \leftarrow F \cup Q\)

16: end if

17: \(R \leftarrow P\)

18: end while

Algorithm 16: Propose-Accept with Backward Radius (PABRT)
Input: \((N, F^0)\), a query network; 
\(k\), maximal query radius; 
\(A_1\), a graph-partitioning algorithm; 
\(p\), number of partitions; 
\(A_2\), a network evaluation algorithm. 
\(A_3\), a partition-matching algorithm. 
Output: \((N, F)\), a fully evaluated query network.

Method:
1: invoke \(A_1\) to partition \((N, F^0)\) into \(p\) subnetworks (s.n.),
- : let \(R\) be the set of s.n., \(\{(N_i, F^0_i)\text{ s.t. } 1 \leq i \leq p\}\);
2: invoke \(A_2\) evaluation on each \(r \in R\);
3: while \(\exists (u, v) \in F^0\text{ s.t. } u \text{ and } v \text{ are in different s.n.}\)
4: \(P \leftarrow A_3(R); //\text{see for example matchPairs}\)
5: for each \((r_i, ..., r_j) \in P //\text{matched merge}\)
6: \(R \leftarrow R \setminus \{r_i, ..., r_j\};\)
7: \(R \leftarrow R \cup \text{merge&eval-a}(r_i, ..., r_j); //\text{see procedure below}\)
8: end for each
9: end while

Algorithm 17: Divide and Conquer for Networks with Acceptance Queries

into one fully evaluated network, using a process resembling PABRT, as appears in Algorithm 18 (it is also possible to recursively call DAC-a on the subnetworks and further partition the graph). DAC-a continues to match and merge these merged networks, until one (minimal) fully evaluated network is constructed.

3.8 Experimentations with Acceptance Queries

We have implemented the presented evaluation algorithms in the same system used for query networks without acceptance queries (see Section 3.5), and again experimented with real and synthetic datasets.

Synthetic Datasets. The datasets are synthesized in the same manner as in Section 3.5, \(\alpha\) and
procedure merge\&eval-a((N_1, F_1),..., (N_n, F_n))
1 : N ← N_1 ∪ ... ∪ N_n; F ← F_1 ∪ ... ∪ F_n;

2 : let C be the set of crossing edges between any two of the n network partitions;

3 : F ← F ∪ C; M ← {n_1 | (n_1, n_2) ∈ C}; // sources of new edges

4 : G ← \{g | g ∈ B(m, l), 1 < l < k; m ∈ M\}; // nodes possibly affected by the addition of crossing edges

5 : for each m ∈ M
6 : evaluate q_p(m), add the result to Q_p;
7 : end for each

8 : return (N, F);

Algorithm 18: Merge and Eval for Networks with Acceptance Queries
\( \beta \) retain their meaning.

**Experiment 1.** In this experiment, we create 5 datasets. The \( i \)-th dataset has \( 5 \times i \) clusters. Each cluster has 180 nodes. \( \alpha = 1/190, \beta = 4 \). One of the following three proposal queries is associated, with equal probability, with each node: (1) Bob’s proposal query from Section 3.6.1; (2) Charlie’s proposal query from Section 3.6.1; (3) A query defining the empty set. One of the following two acceptance queries is associated, with equal probability, with each node: (1) Irene’s acceptance query from Section 3.6.1, without the reference to the attribute; (2) A query accepting all nodes (implemented so as to include only nodes within the maximal backward radius of the accepting node, 3 in this case, since others cannot propose to it). Figure 3.17 shows running time results for algorithms Basic-a, PA and PABRT on these datasets (see next experiment for DAC-a). As expected, the Basic-a evaluation algorithm is the slowest. PA and PABRT perform much better, with a slight advantage to PABRT.

**Experiment 2.** In this experiment, we create 10 datasets according to the same specifications as in experiment 1. Figure 3.18 shows running times results for algorithms PA, PABRT and DAC-a on these datasets. Basic-a is omitted so as not to overload the figure with high numeric values which, similarly to Experiment 1, characterize Basic-a’s run on these datasets. DAC-a is significantly faster than the other algorithms. Here too, we see that PABRT has an advantage over PA.

**Experiment 3.** In this experiment, every set has 160 nodes, \( \alpha = 1/200 \) and \( \beta = 2 \). The rest of the input is as in Experiment 2. Figure 3.19 shows running times results for algorithms PA, PABRT and DAC-a on these datasets. Results for DAC-a are shown as DAC-a1 and DAC-a2. DAC-a1 values are the results when the graph is partitioned according to the synthesized clusters.
Figure 3.18: Results of Experiments 2.

Figure 3.19: Results of Experiments 3.

DAC-a2 values are the results when the Metis partitioner creates the partitions (the number of partitions is equal in both experiments). The results are in line with the results of the previous two experiments. Partitioning with Metis results in typically better running time results than when the partitioning is according to clusters.

Datasets derived from DBLP

Here, too, we also conducted experiments on data derived from the DBLP datasets [28], used to capture patterns of social connections and behavior.

Experiment 4. In this experiment, the graph consists of 1800 nodes, partitioned into 10 parts (using Metis). We create five datasets by gradually lowering $\alpha$ from 55% to 35% in 5% decrements, in order to increase the number of edges in the EDB. Proposal queries are as in Experiments
1 through 3. Acceptance queries are Irene’s acceptance query and a query that accepts nodes whose id is an even integer (i.e., half of the nodes). Figure 3.20 shows the running time results of the four algorithms on these datasets. The results on real datasets are in line with the results on synthetic datasets. The difference between Basic-a and the rest of the algorithms is smaller than in the synthetic datasets, probably due to the relatively dense input graph.

**Experiment 5.** In this experiment, the graph consists of 2000 nodes, partitioned into 16 parts (using Metis). We create five datasets by gradually lowering α from 65% to 55% in 2.5% decrements. The queries remain as in Experiment 4. Figure 3.21 shows the running time results of the PA, PABRT and DAC-a algorithms. The results are similar to the results of the previous experiments.
**Experiment 6.** In this experiment, the graph consists of 2200 nodes, partitioned into 16 parts (using Metis). We create the datasets by gradually lowering $\alpha$ from 65% to 45% in 5% decrements. The queries remain as in Experiments 4 and 5. Figure 3.22 shows the running time results of the PA, PABRT and DAC-a algorithms. The results are similar to the results of the previous experiments.

**Experiment 7.** This experiment examines the influence of the number of partitions on evaluating the network with the DAC-a algorithm. We take the dataset from Experiment 6 in which $\alpha=50\%$. The dataset is partitioned to 6, 12, 18, 24, 30, 36 and 42 partitions. Figure 3.23 shows the running time results. For this dataset, the optimal balance between DAC-a overheads and benefits is between 15 and 20 partitions. Developing a heuristics for finding the optimal number of partitions is left for future research.

**Experiment 8.** This experiment examines the maximum potential benefit NWP from paral-
elizing the evaluation of partitions. For every dataset from Experiment 6, we simulate a parallelized computation with 16 computation units. We measured the longest sequence of partition evaluation, taking into account that evaluating the merge of partitions \( p \) and \( q \) is done after \( p \) and \( q \) are evaluated. Figure 3.24 shows a comparison of the simulated parallel computation and the running time in Experiment 6.

### 3.9 Theoretical Results

We characterize the complexity of problems related to query networks. The following table summarizes the results. The input to our problem contains a query network, which includes data and queries. Therefore, the complexity results are in the sense of combined complexity [93].

<table>
<thead>
<tr>
<th>Problem</th>
<th>Without Acceptance</th>
<th>With Acceptance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Fixpoint</td>
<td>NP-Complete</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>Q. Substitution</td>
<td>P</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>k-Popularity</td>
<td>P</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>Adding Q. Assign.</td>
<td>Not Defined</td>
<td>NP-Complete</td>
</tr>
</tbody>
</table>

#### 3.9.1 The Network Fixpoint Problem.

This Problem is motivated by the need to know (e.g., for optimization) whether more evaluation rounds are needed in order to fully evaluate the network, or if the network is already fully evaluated. Given a query network \( Q \), the problem is to decide whether \( Q \) is fully evaluated.

**Theorem.** This problem is NP-Complete for networks with and without acceptance queries.
Proof. We start with networks without acceptance queries.

NP. A Turing machine can guess which is the node whose evaluation is to yield the addition of an edge. Then, the particular substitution of variables with network nodes, which leads to this addition is guessed. The satisfaction of the predicates is checked, and the input is accepted iff all predicates are satisfied. All steps are polynomial.

The Directed Graph Hamiltonian Cycle (HC) Problem. The HC problem is to decide, given a directed graph $G$, whether there exists a cycle in $G$ that visits each node in $G$ exactly once. The problem is known to be NP-Complete [40]

NP-Hardness. We reduce the directed graph Hamiltonian Cycle (HC) problem to the network fixpoint problem. Consider a HC instance (i.e., a graph) with $n$ nodes. We add to the graph two new nodes, $u$ and $v$. We pick at random a node $a$, and add the edges $(a,u)$ and $(u,v)$. We construct a query network by associating with $a$ a query in the form of a cycle with $n$ nodes (one of which is $a$, since the query is associated with $a$), and a two-edge path starting at $a$ through $u$ to $v$. The query adds an edge between $a$ and the last node in the two-edge path, as illustrated in Figure 3.25. The construction is polynomial in the size of the input. Now,

- If there exists a Hamiltonian cycle in the original instance, $a$ is a part of it, the query is satisfied and the edge is added, i.e., the network was not fully evaluated.

- If there is no Hamiltonian cycle, the query is not satisfied. That is, the network is fully evaluated ($a$ is the only node associated with a query that may add an edge).

The problem is therefore NP-Hard.

With Acceptance Queries. The problem remains NP-Complete for the model with acceptance queries. In this case, the network fixpoint instance includes, for every node, an acceptance query which accepts all proposals.

3.9.2 The Query Substitution Problem

This Problem is motivated by the need to know whether a participant interested in creating a connection with a particular participant $p$ can change her proposal policy so that after evaluation, a connection to $p$ is created. The problem is to decide, given a fully evaluated network and two
nodes $u$ and $v$ in the network, whether there exists a query $q_u$ such that if we replace $u$’s original proposal query by $q_u$, and fully evaluate the resulting network, then the edge $(u,v)$ is added.

**Theorem - without acceptance queries.** The problem is in $P$ for the model without acceptance queries.

- If $v$ is reachable from $u$ then $q_u$ exists, and it is the query which proposes a connection to every friend of a friend of $u$ (the foaf query).
- If $v$ is not reachable from $u$, then $q_u$ does not exist.

The complexity is therefore equivalent to the complexity of deciding whether $v$ is reachable from $u$, which is trivially polynomial.

**Theorem - with acceptance queries.** In the presence of acceptance queries, the query substitution problem becomes NP-Complete.

**Proof.**

**NP.** The number of predicates in a satisfiable query is at most $2|N|^2$, which is the maximum number of edges in the network (assuming no predicate repetitions in the query). A Turing machine can guess the query. Then, guess a series of edge additions which leads to the addition of $(u,v)$. The guess includes the substitution which satisfies each proposal and acceptance query related to the addition of the edge. If all substitutions indeed satisfy the corresponding queries, the input is accepted. Otherwise, rejected. Thus, if there exists a query $q_u$, then there is a guess that leads to
the acceptance of the input.

**NP-Hardness.** We reduce the HC problem to the Query substitution problem. Consider an HC instance \((G)\), in which the graph has \(n\) nodes.

We polynomially construct a query substitution instance. We add the following nodes to \(G\):

- Three new nodes, \(u, u', v\).
- Two new edges, \((u,u')\) and \((u',v)\).
- An edge from a random node \(a\) in \(G\) to \(u\).

See Figure 3.26 for an illustration of this construction. Node \(v\)'s acceptance query is illustrated in Figure 3.26. The rest of the proposal and acceptance queries in the network are unsatisfiable. The network is therefore fully evaluated. Nodes \(u\) and \(v\) have the same roles as \(u\) and \(v\) in the problem definition. The problem is therefore to decide whether there exists a query \(q_u\) s.t. if \(u\)'s original (unsatisfiable) proposal query is replaced by \(q_u\), and the network is fully evaluated, then the edge \((u,v)\) is added. Now,

- If there is an HC in \(G\), \(u\)'s unsatisfiable proposal query can be replaced by the foaf query.
  In this case, \((u,v)\) is proposed, \(v\)'s acceptance query is satisfied, and \((u,v)\) is added. A query substitution therefore exists.
- If there is no HC in \(G\), \((u,v)\) cannot be added as a result of any substitution for \(u\)'s proposal query. Therefore, a query substitution does not exist.

Thus, the problem is NP-Hard. ■

### 3.9.3 The k-Popularity Problem.

The *k-Popularity Problem* is motivated by the need to know whether a certain proposal query could result in the creation of a certain number of connections. This could be in order to avoid creating more (or, less) connections than desired. The problem is to decide, given a query network, a node \(u\) and an integer \(k\), whether there exists a proposal query for \(u\), \(q_u\), such that after full evaluation,
Figure 3.26: The query substitution instance for the corresponding reduction.

Figure 3.27: The k-Popularity instance for the corresponding reduction.
$u$ has at least $k$ connections which are accepted proposals made by $q_u$ (i.e., $k$ 'new' connections that are not in $F^0$).

**Theorem - without acceptance queries.** The problem is in $P$ for the model without acceptance queries. If the number of nodes reachable from $u$ is $k$ or more, then $q_u$ exists and it is the foaf query. Otherwise, no such query exists.

**Theorem - with acceptance queries.** The same problem, but in the model with acceptance queries, is NP-Complete:

**NP.** A Turing machine can guess $k$ nodes to which $u$ could be connected in the fully evaluated network. Then, the machine can guess $k$ series of $O(|N|^2)$ edges which lead to the addition of these connections. The guesses can be verified using the polynomial process described in 3.9.2. The problem is therefore in NP.

**NP-Hardness.** We reduce the HC problem to the k-Popularity problem. Consider an HC instance with graph $G$.

We polynomially construct a k-Popularity instance:

- $k$ is the number of nodes in $G$.
- We add two nodes, $u$ and $v$, to $G$.
- We add the edges $(u,v)$ and an edge from $v$ to each node in $G$.

We associate a foaf proposal query with $u$, and associate with each node of $G$ (but not to $u$ and $v) an acceptance query that accepts a node if it is connected through a two-edge path to a member of a cycle of size $k$, and if the accepting node itself is a member of the cycle. The rest of the queries are unsatisfiable. This k-Popularity problem instance is illustrated in Figure 3.27. Now,

- If there is an HC in the original graph $G$, then in the fully evaluated network of the constructed k-Popularity instance, $u$ is added as a connection to the $k$ members of the cycle. Therefore, a query which leads to the creation of least $k$ connections of the form $(u,\cdot)$.
- If there is no HC in the original graph $G$, then for any proposal query for $u$, the fully evaluated network of the constructed k-Popularity instance does not have new edges (i.e., edges which did not exist in the constructed k-Popularity instance).

The problem is therefore NP-Hard. ■
3.9.4 The Edge-Creating Query Assigning Problem.

The following problem (as well as the next problem) may have a role in contexts related to automatic creation of social networks with proposal and acceptance queries, for benchmarking or spam detection. Consider (1) a network; (2) two nodes in the network, \( u \) and \( v \); (3) an acceptance query \( q_a \); (4) a proposal query \( q_p \). Let \( q_{au} \) be an acceptance query which accepts all the nodes in the network and let \( q_{p\emptyset} \) be an unsatisfiable proposal query. Let \( Q_a \) be \( \{q_a, q_{au}\} \) and let \( Q_p \) be \( \{q_p, q_{p\emptyset}\} \). The Edge-Creating Query Assigning Problem is to decide whether there exists a function \( f : N \rightarrow Q_a \times Q_p \) such that in the network resulting from associating each node \( n \) with the proposal query and the acceptance query in \( f(n) \), and after full evaluation, the edge \( (u,v) \) exists.

**Theorem.** The problem is NP-Complete.

**Proof.**

**NP.** Guess \( f \). Then, a series of edge additions leading to the addition of \( (u,v) \) is guessed and verified using the polynomial process described in Section 3.9.2.

**NP-Hardness.** We reduce the directed graph Hamiltonian Cycle (HC) problem to our problem. Consider a HC instance with \( n \) nodes. We take an arbitrary node \( w \) and split it into two nodes, \( u \) and \( v \). Edges of the form \( (w, \cdot) \) become \( (u, \cdot) \). Edges of the form \( (\cdot, w) \) become \( (\cdot, v) \). Let \( q_p \) be a proposal query which proposes a connection to nodes at the end of a path of length \( n \) edges. Let \( q_a \) be \( q_{au} \). The construction is polynomial.

- If there exists a Hamiltonian cycle before the split, then a function \( f \) which assigns all the nodes with \( q_{au} \) and \( q_p \) is to result in the addition of \( (u,v) \), since after the split, \( u \) and \( v \) have a path of length \( n \) edges between them.

- If there is no Hamiltonian cycle before the split, then all the existing cycles have less than \( n \) nodes. Assume to the contrary that edge \( (u,v) \) is added. According to the preservation of cycles property proved in Section 3.3.1, network evaluation cannot result in cycles that are larger than cycles which existed before the evaluation. However, if \( (u,v) \) is added, then in the network before the split, there existed a cycle of size \( n \), a contradiction.

The problem is therefore NP-complete. ■
3.9.5 Model Variants

The Dunbar number [32, 33] is defined as the maximal number of human connections that an individual can effectively and simultaneously maintain. In the model presented above, there is no consideration of such a parameter. A node can potentially have a connection to each of the nodes in the network. Next, we consider alternatives for model semantics which include the concept of a Dunbar number. Let $d$ be the maximal allowed number of edges emanating from a node in the network.

**First Come First Served.** According to this semantics, edges outgoing from a node $u$ are added as long as the total number of edges outgoing from $u$ does not exceed $d$. If a node has $d$ connections, no more connections can be added to it. If a node $n$ has $d - m$ connections and evaluating $n$ is to yield more than $m$ edges, then only $m$ new edges are added, say to the minimal $m$ nodes according to some total order on $N$. Note that according to this semantics, the network always reaches a fixpoint, but more than one fixpoint is possible. The particular fixpoint depends on the order of node evaluations.

**Random.** According to this semantics, if a new edge is to be added to a node $u$, and $u$ already has $d$ connections, then a random existing edge is chosen and deleted before the new edge is added (the deleted edge could be used to satisfy queries). A network with this semantics does not necessarily reach a fixpoint after a finite number of evaluation rounds.

**First In First Out.** This semantics (FIFO) assumes that each edge in $F^0$ has a unique id, and that there is a total order $\prec$ over these ids. Every edge added to $F$ is assigned a new and unique id which is greater, according to $\prec$, than all the ids in $F$. If a node with $m_1$ connections is to be added $m_2$ connections, then the $m_1 + m_2 - d$ connections of this node that are minimal according to $\prec$, are deleted before the addition. Note that when $d = 1$, FIFO and Random semantics coincide. A network with FIFO also does not necessarily reach a fixpoint. In Figure 3.28, we show a network with the FIFO semantics in which $d = 2$, that oscillates and does not reach a fixpoint (which is therefore an oscillatory example for the random semantics as well). In the example, all nodes accept all nodes. The first evaluation of $u$’s proposal therefore adds the edges $(u,v_1)$ and $(u,w_1)$, and deletes $(u,v_0)$ and $(u,w_0)$. Let $i \% j$ denote $i$ modulo $j$. Every subsequent evaluation adds $(u,v_{i(\%3)}$ and $(u,w_{i(\%3})$ and deletes $(u,v_{(i-1)(\%3)}$ and $(u,w_{(i-1)(\%3)}$ for some $i$. A fixpoint is not reached.

**Priority-based.** According to this semantics, edges have weights that represent their priority.
Weights are on a finite scale of, say 1 to 100. The proposal query defines the initial weight with which the edge is added. This weight depends (among other possible factors) on the node to which the connection is added. If a node has $d$ connections, a new connection with weight $w$ can be added only if $w$ is heavier than an existing weight. One of the edges whose weight is less than $w$ is deleted, according to a policy (e.g., the minimal). A network with this semantics necessarily reaches a fixpoint, since the sum of all weights $(1)$ can only increase and $(2)$ is bounded.

### 3.10 Related Work

The structure and growth patterns of social networks are topics that have been studied in many contexts, from physics and biology related systems to transportation, telephony and internet networks [58]. [58] also reviews several models for network growth. Many recently published works in the area of social networks are concerned with analysis of social networks structure, privacy and security in social networks, search related issues and many more. In SIGMOD record, March 2008, as part of the report on the Databases and Web 2.0 Panel at VLDB 2007 [9], it is stated that "Understanding and analyzing trust, authority, authenticity, and other quality measures in social networks will pose major research challenges." In fact, adding queries, or programs, to nodes can be an extension for these topics to the actual running of a technology-assisted social network.

Historically, the evolution of information and structure in social networks has been a subject of research for more than a hundred years. See [101] for an overview of the subject. In particular, the interactive applet at [67] provides an interesting view as to the inner workings of such networks. Of
course, the Internet has brought social networks to the forefront. Perhaps, the most distinct feature of these networks is that they are centrally managed by and relatively open to new participants. It is not clear whether parameters such as Dunbar’s number [32, 33], a limit on the number of individuals with whom a person can consistently interact over time (believed to be around 150), is still valid in the WWW context. Conceivably, technology can assist in marinating more meaningful connections or even many weak but useful connections.

Related work in the area of Datalog evaluation and optimization is thoroughly covered in [92] and [2]. However, these assume small queries and large data. Processing of Web and Internet data with Datalog-like formalisms is studied in [14], [4], [78], [64]. We are not aware of work regarding very large queries or regarding Datalog with a massive number of rules.

Models for strategic network formation are covered in [53]. Unlike our model, these models are not query-based. For example, the strategic model in [12], is based on payoffs that participants receive by forming connections.

SoQL [80] is an SQL-like language for social networks. It includes language constructs which support typical information needs for social networks. In the context of this work, such queries may provide a more expressive alternative to the conjunctive queries of this paper.
Chapter 4

Protocols for Social Networks

4.1 Introduction

Online social networks, such as Facebook [35], MySpace [70], Twitter [89] and LinkedIn [62], have gained unprecedented popularity. The number of users and connections in social networks has grown dramatically, as the networks proliferate to new target audiences and add new features.

This popularity has brought very large amounts of data into social networks [6], which are difficult and expensive to manage manually. Even the data available to a single user has become hard to manage manually, and automation features are being developed (e.g., [36], [91], to be discussed again below).

Nielsen’s report from August 2010 [72] shows that the monthly time spent by internet users on social networks is more than the time spent on e-mail, search, instant messaging, portals and classifieds combined. Increasing the productivity of social networks users is therefore a pressing challenge motivating automation, in particular since social networks are widely used in many business contexts. Indeed, efforts are put into automating social networks ( [36], [91], [80], [83], [84]). These, however, do not include communication between users.

Another type of social network automation are applications which monitor the public information in social networks, for purposes such as brand monitoring (e.g., [75], [95] and many others). In addition, such technologies demonstrate preliminary efforts to engage in social networks, rather than passively monitor. Automating protocols, that we discuss in this work, helps participants engage in the social network in an efficient manner.
In order to make such applications reliable decision support tools available around the clock, which can be more useful for social needs and effective in business contexts, it is necessary to let users safely rely on others’ decisions, automate responses, verify the consistency of decisions with possible user requirements and modify decisions as situations in the network develop. Moreover, the distributed nature of activities in social networks necessitates that different decision making processes be able to operate concurrently without interfering (and even in synergy) with each other.

In this chapter, we study automation in social networks using protocols. First, we introduce a generic social protocol which addresses the basic problem of making a determination of status regarding an entity in the social network. The protocol can be modified into a wide variety of concrete protocols corresponding to likely usages, using simple building blocks such as deadlines and forwarding privileges. Then, we develop protocols which enable changes in statuses, by ensuring that sets of changes are performed atomically, and therefore always preserve network consistency (to be defined later on).

We start with protocols for a Twitter-like communication architecture, in which messages can only be sent from participants to their followers (One-Way communication architecture, to be defined). We continue with a more liberal Two-Way communication architecture approach. The protocols enable atomic, consistent changes of status in the social network. Multiple protocol instances can operate concurrently. In particular, protocol instances are able to take advantage of work done by other protocol instances.

By this, we in fact lift the traditional concurrency control isolation requirement (see [19], [99]), and show that a more liberal communication architecture contributes to concurrency. Instances can also mutually help each other succeed.

### 4.1.1 Directed Networks and Communication

Some social networks, such as Facebook, use an undirected graph model. Others, such as Twitter [89] and Google Buzz [43], use a more general directed model. We maintain that social networks are in fact directed graphs (with many reciprocal connections). Social networks already demonstrate that there are many types of inherently directed connections, such as the Facebook ‘like’ connection. We therefore use the more general, directed graph-based formalism in our net-
work model. Adopting the Twitter terminology, if a (directed) edge \((u,v)\) exists in the network, we say that \(v\) follows \(u\).

We consider two communication architectures in the directed social network model. First, we consider the \textit{one-way} communication architecture. According to this architecture, a participant \(v\) can send messages to any participant \(u\), provided that an edge \((v,u)\) exists. This policy is used, e.g., in Twitter.

Then, we consider the \textit{two-way} communication architecture. According to this architecture, messages can be initiated as in the one-way architecture, i.e., from \(v\) to \(u\). However, a response to such a message may be sent from \(u\) to \(v\) as well. The rationale in allowing such responses is that in the context of a protocol instance, if \(v\) sent a message to \(u\), \(v\) is interested in \(u\)’s decisions in the protocol.

### 4.1.2 Chapter Outline

In Section 4.2, we present a generic protocol for initial determination of status in a social network. Modifiers shape the behavior of the generic protocol. In Section 4.3, we develop methods for consistency-preserving installment of status changes in the network, for directed networks with a one-way architecture. In Section 4.4, we study the same problems for the two-way communication architecture. In Section 4.5, a proof-of-concept implementation is presented. In Section 4.6, we characterize the complexity of several related theoretical problems.

### 4.2 A Protocol for Status Determination

We introduce a protocol which automates the determination of status regarding an event in the social network, such as a party, a commercial offer, a demonstration etc. The protocol coordinates the decisions, which are taken considering the participants’ social consistency requirements, which depend on decisions made by their friends. As in the real world, these requirements are such that a participant \(p\) can make a consistent initial decision even if only some (or none) of the participants that \(p\) follows made a decision. A protocol instance is started by an \textit{initiator} and unlike protocols discussed in Sections 4.3 and 4.4 does not commit or abort, since every initial status decision is consistent. Here, we use a two-way communication architecture.
Next, we give examples for likely usages of such protocols.

4.2.1 Examples

Consider a university reunion event. Bob, the organizer, would like to use the social network in order to organize the event, according to the following rules:

- There are two types of invitees. Alumni, and Bob’s personal friends, who are not alumni.
- Alumni may forward the invitation to other alumni. Others may not.
- Alumni who would like to attend need to RSVP at least two days before the event.
- $k$ sleep-over beds are reserved by Bob. These will be allocated, by the protocol, to the first $k$ interested guests.

The protocol that we introduce automates the coordination of events (such as this reunion). We use a generic protocol and modifiers, which are programming-like, and sometimes query-based, features which tune the generic protocol to fit particular needs, such as Bob’s needs.

The following example is the first message of a protocol instance, using which $p$ organizes a dinner with $k$ places, and the invitees determine their status (true or false) regarding the dinner. The invitation is to participants distant at most two edges from $p$ (say, satisfying query $q_2$). Each initial invitee receives $k'$ (the sum of these $k'$ has to be less than $k$) places, and may take up to two places. These conditions are conveyed in a structured message that shapes the possible subsequent protocol messages, which coordinate event. The semantics is to be defined. The message is:

$$\text{(Event=dinner; Type=1; Forwarding=(q_2, unlimited); Feedback=(p, true); Resource=(k', 2))}$$

4.2.2 Protocol Messages

As mentioned in the example, the protocol is designed as a generic protocol, together with a set of modifiers that can control its behavior. The generic protocol has three types of messages:

- **Type 1: Initiation Message.** This type of message is sent only by the participant that starts the protocol instance (the initiator) to some of, or all of, its followers, and is always the first message related to an entity (an event, commercial offer, etc.).
• **Type 2: Forwarding Message.** This type of message may be sent by a participant, who received a type 1 or 2 message in a protocol instance, to its followers.

• **Type 3: Feedback Message.** Feedback messages may be sent after the receipt of a message \( m \) or types 1 or 2. For simplicity, we assume that feedback is tweeted with a tag, and that the participant who required feedback follows messages with the tag regardless of its source (common Twitter practices). A less attractive, but feasible, option is to use the two-way communication architecture and have feedback messages passed from one participant to the other along the path, up to the addressee.

### 4.2.3 Modifiers

Next, we define the modifiers that can be applied to the generic protocol.

**Forwarding Privileges.** According to the generic protocol presented above, participants can freely send Type 2 messages. In many cases however, initiators (as well as others) are likely to want to limit this ability. The *Forwarding* modifier provides control on the privilege to forward protocol messages. A forwarding privilege is composed of two parameters. The first controls to whom the message can be forwarded. The second controls whether there is a permission to forward further or not. Table 4.1 specifies the possible values for forwarding privileges (the *Forwarding* modifier).

**Example.** In Bob’s protocol from Section 4.2.1, alumni have forwarding privilege \((q,\text{unlimited})\), where \(q\) is satisfied only by alumni. Personal friends will receive forwarding privilege \(false\).

**RSVP.** A participant who sends a message, either as an initiator or as a forwarder, may require a feedback message from the recipient when the recipient changes status. For this, we use feedback messages (Type 3). A feedback requirement may be such that only participants satisfying a query \(q\) are required to send feedback. Feedback messages are sent to the participant who required them. Table 4.2 specifies the possible values for the *Feedback* modifier.

**Deadlines.** A message with a feedback requirement may have a deadline, by which the receiver has to determine a status w.r.t. the entity under discussion. The *Deadline* modifier’s value is this point in time. If this time has passed before status was determined, the participant cannot determine
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>No forwarding allowed.</td>
</tr>
<tr>
<td>true, unlimited</td>
<td>Forwarding allowed, with permission to forward further.</td>
</tr>
<tr>
<td>true, limited</td>
<td>Forwarding allowed, without permission to forward further.</td>
</tr>
</tbody>
</table>
| query, unlimited | · Forwarding allowed  
|               | · Only to those satisfying the query  
|               | · With permission to forward further. |
| query, limited   | · Forwarding allowed  
|               | · Only to those satisfying the query  
|               | · Without permission to forward further. |

Table 4.1: Forwarding Privileges

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>No feedback required on change of status.</td>
</tr>
<tr>
<td>query, $p$</td>
<td>Feedback to $p$ is required on change of status only from participants satisfying the query.</td>
</tr>
<tr>
<td>true, $p$</td>
<td>Feedback to $p$ is required on change of status.</td>
</tr>
</tbody>
</table>

Table 4.2: Feedback

status w.r.t. this entity. Note that the algorithms in Sections 4.3 and 4.4 in fact work on parts of the social network in which all participants determined status.

**Example.** In Bob’s protocol, the feedback modifier is $(true, Bob)$, and the value for the deadline modifier is set to two days before the event.

**Using Resources.** Protocols may have the functionality of letting participants use a communal resource (such as tickets, carpool seats etc.). A resource can also be negative (debt, tasks, etc.). The Resource modifier is a two-tuple $(k, l)$ where $k$ is the total amount of available resource ‘allocated’
to the recipient and $l$ is the maximum amount that one participant can take. For example, a limit of two may be appropriate for events where participants are likely to show up with their spouses. If participants do not exhaust a resource and do not want to forward it, they may return the resource by sending a Type 3 message to any participant on the path from the initiator to themselves, including the amount of resource returned. Note that modifying the protocol to handle a resource can be in conjunction with a regular feedback requirement. Feedback messages related to the resource return resources to one of the forwarders (indicating the returned amount). Feedback messages related to RSVP requirements are sent, as usual, on a status change.

**Example.** In Bob’s protocol, there are $k$ sleepover places. The first message from Bob to recipient $i$ has resource modifier $(k_i, 1)$, where the sum of $k_i$ is $k$, the number of available sleepover places. Each participant can take up to 1 place.

### 4.2.4 Politeness

Effective usage of protocols requires that participants do not forward a privilege or a resource that is not available. Privileges and resources can therefore not be increased when forwarded. Abidance by this is called *politeness*. Next, we discuss how politeness is reflected in view of the modifiers.

**Forwarding Politeness.** A participant $p$ who is allowed to forward cannot give forwarding privileges that $p$ does not have. In general, $p$ may replace *unlimited* by *limited*, *true* by *query* or *false*, and *query* by *false*. In addition, $p$ could replace a privilege with query $q$ by a query $q'$ where $q'$ is *contained* in $q$ (see [92] for definition of query containment).

**RSVP Politeness.** A forwarder, say $p$, of a message without a feedback requirement may add a feedback requirement with $p$ as the feedback recipient. If a message includes a feedback requirement (say, to $q$) $p$ cannot lift the requirement, but may replace $q$ by $p$ as the feedback recipient. In this case, $p$ is responsible for the feedback to $q$, and the next recipient is responsible for the feedback to $p$. Here too, forwarders may replace a query $q$ with $q'$, if $q$ is contained in $q'$ (i.e., $q'$ forces ‘more’ feedbacks).

**Deadline Politeness.** A forwarder may add a deadline to a protocol message which was re-
ceived without a deadline. If such a message already has a deadline $d$, the forwarder may set a deadline which is not later than $d$.

**Resource Politeness.** A participant receiving a message with a resource modifier value of $(k, l)$ may forward the message with resource values $(k', l')$ where $k' \leq k$ and $l' \leq l$.

### 4.2.5 Structure of Messages

A message is a set of attributes with values. The attributes and values are:

- **Entity** - the id of the entity regarding which statuses are being determined (e.g., an event). It must appear in every message.
- **Type** - the number of type of the message (1, 2 or 3, as defined in Section 4.2.2).
- **Forwarding** - the forwarding privilege, as defined in Table 4.1.
- **Feedback** - the feedback requirement, as defined in Table 4.2.
- **Resource** - in type 1 or 2, the resource available as defined in Section 4.2.3.
- **Deadline** - a point in time by which status has to be determined.
- **Status** - in a type 3 message only, the determined status.

Many examples of messages can be found in the next section.

### 4.2.6 Usage Examples

We demonstrate the usefulness of these methods by providing examples for the range of possibilities that they encompass. Entities in the social network are the objects regarding which participants make decisions (typically events). Their ids are henceforth denoted as $e_i$. In the following examples, we give values to modifiers and briefly discuss the resulting protocol’s properties.

**Meeting for two.** $p$ sends the following initiation message to a single participant in order to arrange a meeting $e_0$ only for the two participants. A type 3 message with an RSVP is required when the invitee changes status w.r.t to $e_0$:

```
(Entity=\text{e}_0; \ Type=1; \ Forwarding=\text{false};
\ Feedback=(\text{true}, p))
```
**Meeting with deadline.** Here, $p$ sends a non-forwardable message to a few participants, with a deadline requirement:

$\text{(Entity=e}_1; \text{ Type}=1; \text{ Forwarding}=false; \text{ Feedback}=(true,p); \text{ Deadline}=12:30)$

Setting $\text{Forwarding}=(true,unlimited)$ would give the invitee an option to forward the message to anyone.

**A surprise party.** $p_1$ would like to organize a surprise party, $e_3$, for $p_2$. Participants must be direct friends of $p_2$, and $p_2$ is by no means to become aware of the party. Let $q_1$ be a query satisfied by participants directly connected to $p_2$ (and are not $p_2$). The following is the Type 1 message which starts the protocol. No RSVP necessary.

$\text{(Entity=e}_3; \text{ Type}=1; \text{ Forwarding}=(q_1,unlimited); \text{ Feedback}=false)$

**An open party.** This party $e_4$ is open to everyone:

$\text{(Entity=e}_4; \text{ Type}=1; \text{ Forwarding}=(true,unlimited); \text{ Feedback}=false)$

**Tickets Giveaway.** This message is for a ticket giveaway, in which $p$, the initiator, gives away 3 tickets to every message recipient. Every interested participant can take up to 1 ticket. Forwarding is allowed, RSVP not required:

$\text{(Entity=e}_5; \text{ Type}=1; \text{ Forwarding}=(true,unlimited); \text{ Feedback}=false; \text{ Resource}=(3,1))$

A recipient of such a message, who took one ticket and would like to give the other two to two friends, may send each friend the following subsequent Type 2 message (which respects the politeness rules):

$\text{(Entity=e}_5; \text{ Type}=2; \text{ Forwarding}=false; \text{ Feedback}=false; \text{ Resource}=(1,1))$

A feedback message for returning two tickets is:

$\text{(Event=e}_5; \text{ Type}=3 \text{ to } p; \text{ Resource}=2)$
A dinner party with k places. $p$ organizes a dinner, $e_6$, with $k$ places. The invitation is to participants distant at most two edges from $p$ (say, satisfying $q_2$). Each initial invitee receives $k'$ places, and may take up to two places. Note that a participant need not necessarily give the second place to someone who satisfies $q_2$:

$$(\text{Event}=e_6; \text{Type}=1; \text{Forwarding}=(q_2,\text{unlimited}); \text{Feedback}=(p,\text{true}); \text{Resource}=(k',2))$$

4.2.7 Prototype Implementation

The generic protocols are implemented using Twitter’s API (through Twitter4J [90]) called from a Java code which implements protocol logics. We operate on real Twitter accounts that we created, interconnected as a network (a concept also used in [81]).

Upon the receipt of a protocol message, the participant is provided a GUI using which they can forward the message, according to the values of the modifiers, as well as set their status w.r.t the event. The protocols logic is enforced.

Example. Marge organizes an academic workshop with 12 participants. She uses her social network in order to distribute the invitations (but without exceeding 12). She invites Bart and Lisa, and gives each 6 ‘tickets’ to the workshop, with permission to forward them, and a deadline set to July 15th (Figure 4.1). Bart decides to attend, tweets his decision on Twitter, takes a ticket, and forwards 5 tickets with the right to forward and a deadline set to July 10th (Figure 4.2). Lisa tweets a negative decision, forwards one ticket without the right to forward further, and returns 5. A Twitter account following the protocol’s participants appears in Figure 4.3. The protocol
continues until the resource is exhausted.

4.3 Protocols for Consistency-Preserving Changes

In the previous section, we discussed initial determination of statuses. We now consider another situation, in which statuses are already determined, and consistent with the network participants’ requirements. We develop a model and protocols which allow a participant to coordinate changes in status with other participants. The coordination assures that the network remains consistent.
4.3.1 Model

Here too, given an edge \((u,v)\), we use the Twitter terminology and say that \(v\) follows \(u\). In the model, each participant makes decisions based on the decisions made by his/her followed participants. These decisions that participants make are decisions regarding \(events\) that participants disperse in the network. In the spirit of ‘Facebook Events’, we restrict decision values to be \texttt{true} or \texttt{false}. The algorithms we present may be extended to a larger finite set of values beyond just two. These values are determined as an event \(e\) is dispersed in the network, and participants make decisions in accordance with their \textit{consistency requirements} (for example, using the protocols described in the previous section).

Decisions are made in accordance with \textit{consistency tables}, which are a simple abstraction of the conditions under which participants will determine a \texttt{true} or \texttt{false} status. In a real social network, such rules are complicated, private and may change with time, and need not be defined by the participants (rules could be acquired using profiles, past behavior, an intuitive GUI, etc.). This gives further motivation to the protocols approach, in which coordination is based on questions and answers, rather than methods which assume that the whole data is available.

Later on, we prove that several underlying problems are NP-Complete. In the context of large networks (of say, a couple dozen or more participants), this intractability result is yet another motivation to use the protocols approach as a heuristics for the problem.

For simplicity, we assume that decisions related to distinct entities (e.g., events) in the social network are not dependent on each other. Therefore, the discussion is restricted to networks in which only one such entity exists.

The model that we use is \textit{interpreted}. We use the values that protocol instances read and write, and our methods take advantage of this knowledge. This is in contrast to the traditional database transaction model (thoroughly covered in [19] and [99]), which is generally \textit{uninterpreted}, and is oblivious to the semantics of the operations within the transaction. Semantic factors influence the ability to provide concurrency, and mutual help, among protocol instances.

In this section, we investigate problems arising from the need to provide atomic and consistent status changes in the social network, as demonstrated in the following examples.
4.3.2 Examples

Consider the network in Figure 4.4. Participants have status values (true or false), which reflect their decision to attend or not attend a party. The figure also lists conditions (on followed participants, as discussed above) under which Homer, Lisa, Bart and Cynthia will attend the party. The statuses in the network are consistent with these requirements. Consider the following chain of events:

- Marge would like to change her mind, and not go to the party. Change in her status may affect Bart’s and Lisa’s consistency.
- Therefore, before the change takes effect, Marge notifies Bart and Lisa that she wants to change status.
- Lisa does not approve Marge’s change.
- Bart, on the other hand, may approve the change, if he can change status to true. He notifies his followers, Homer and Lisa.
- Homer approves Bart’s change (without having to change his own status).
- Assuming that Bart changes status to true, Lisa can approve Marge’s and Bart’s changes.

If all these changes (i.e., toggling Bart’s and Marge’s statuses) are installed, the system is transformed to a new, consistent state (this is scenario 1). If only one of them is installed, the network becomes inconsistent.

In order to enable consistency-preserving, atomic changes, we use protocols, which coordinate the changes under the one-way architecture requirements. Obviously, the rationale is not to force a participant to go to a party, but rather to help the participants remain consistent as a group. In some organizations, one can actually impose atomic, consistent changes. In other settings, this may be done on a voluntary basis, and/or as a result of mutual interests.

Aborting a protocol instance. Let us return to the original statuses as shown in Figure 4.4. Suppose that Homer would like to change status to false. He notifies his only follower, Cynthia. Cynthia approves only if she can change status to false, and therefore notifies her followers, Lisa and Homer. Homer approves. Lisa does not approve. The attempt is thus concluded without transforming the network to a new consistent state (this is scenario 2). None of the statuses are changed.
• Homer goes iff Bart or Cynthia or both go

• Lisa goes iff Bart goes or both Cynthia and Marge go

• Bart goes iff Marge does not go

• Cynthia goes iff Homer goes

Dashed arrows point to values after scenario 1

Figure 4.4: Social Network for scenarios 1 and 2.

<table>
<thead>
<tr>
<th>Bob</th>
<th>Alice</th>
<th>Jerry</th>
<th>Garfield</th>
<th>Simba</th>
<th>Tom</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bob</th>
<th>Garfield</th>
<th>Simba</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tom</th>
<th>Alice</th>
<th>Jerry</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Figure 4.5: Social Network for scenario 3. A table belongs to the participant whose column is grey.

We call this process an *abort* of a protocol instance.

**Concurrency and Mutual Help.** If scenario 1 were completed before Homer’s regret (of scenario 2), Lisa could approve Cynthia’s change to *false*. The methods we develop later on allow such protocol instances to operate concurrently without having to wait for each other’s conclusion. More importantly, sometimes protocol instances that cannot succeed on their own, are able to work in synergy so as to transform the network to a new consistent state, as demonstrated next.

Consider the network and tables in Figure 4.5. The tables list all the consistent states for each participant. Suppose that:

- Bob wants to change status (to *true*). He notifies Simba, who approves the change only if he can change status (to *true*).
- Simba notifies Tom, who also needs to change status (to *true*).
- Tom notifies Jerry, who does not approve the change, and the protocol instance, henceforth *PI*,
In addition:

- Alice wants to change status (to true). She notifies Jerry, who can approve if he can change status (to false).
- Jerry notifies Garfield, who can approve if he can change status (to false).
- Garfield notifies Simba, who does not approve the change, and this protocol instance, say P2, aborts.

However, if P1 and P2 operated concurrently and made assumptions on the success of each other, both would succeed. Consider a situation in which at a certain point P1 operated until just before Tom notifies Jerry, and P2 operated until just before Garfield notifies Simba, as illustrated in Figure 4.6(a). Now, P1 continues and Tom notifies Jerry. Jerry could approve Tom’s change (to true) if P1 succeeds, since in this case Alice is in status true. P2 continues, and Garfield notifies Simba (i.e., Garfield wants to change to false). Simba could approve Garfield’s change, but only if P1 succeeds, since in this case Bob is in status true.

At this point, installment of the changes of both P1 and P2 results in a consistent state, i.e., the state created from toggling all statuses in the network (this is scenario 3). We shall model such ‘success’ dependencies in order to allow concurrency as well as take advantage of opportunities which arise from such synergy between protocol instances.

Figure 4.6: (a) P1 and P2 just before operating on the same nodes. (b) CAG discussed in Section 4.3.5

aborts.
4.3.3 Formalization

Social Network. A social network is a directed graph \((V, E)\). \(V\) is a set of nodes, representing network participants. \(E\) is a set of directed edges, representing connections between participants. Given a participant \(p\), the input for \(p\)’s decisions are decisions made by participants \(q\) s.t. \((q, p) \in E\) (i.e., participants that \(p\) follows). As mentioned in Section 3.1, participants determine their status w.r.t. events. As discussed above, we assume that decision regarding distinct events do not depend on each other, and initially restrict the discussion to the case where only one event, \(e\), exists in the network. We relegate treatment of interdependent events to future work.

Statuses. Every \(p \in V\) is associated with a status value, which is \(p\)’s status w.r.t. \(e\). Status values are \(\{\text{true, false}\}\).

Consistency Tables. Each \(p \in V\) is associated with a table \(T_p\). \(T_p\) has an attribute for each participant that \(p\) follows (i.e., for each incoming edge), and an attribute for \(p\). \(T_p\) lists \(p\)’s possible status decisions, for combinations of true/false statuses of participants that \(p\) follows. We say that \(p\) is in consistent state if \(T_p\) contains a tuple which corresponds to \(p\)’s status and to the statuses of participants that \(p\) follows. Not all possible combinations of followed participants’ decisions have to appear in the table.

Network State and Consistency. A network state \(S\) is a function \(S : V \to \{\text{true, false}\}\). Given a network and a state \(S\), we say that the network is consistent (in state \(S\)) if all participants are in consistent states. That is, if \(S\) corresponds to a tuple in \(\bigtimes_{v \in V}(T_v)\). Combinations of statuses which do not appear in \(T_p\) for some participant \(p\) can therefore not occur in a consistent state.

Although we define consistency using a join operation, it is impractical to compute this join, since (1) tables are socially sensitive and are therefore private; (2) even if tables were available, materializing this join is impractical since merely a query defining the join is in the order of magnitude of the data being queried (the number of tables in the query is up to the number of nodes in the network) and (3) the result of such a join would typically be extremely large.

As discussed in Section 3.1.2, these tables are a simplification of what a person would do, or an abstraction of the actions of an automated tool, if decisions are made automatically.
Example. The network in Figure 4.5 is in a consistent state, corresponding to a tuple in the join of all tables. If, e.g., only Alice toggles status, Jerry is inconsistent. If all participants toggle status, the network is once again consistent.

Communication Architecture. Our communication architecture is One-Way, in which information is sent from a node $v$ only to $v$’s followers. This is the situation in many social networks, e.g., Twitter and Google Buzz.

The System. The system administers the protocols, and can send and receive messages/updates to and from every node. The system can work in one of two modes which affect its scalability to a large number of protocol instances:

- Unlimited memory, in which the system can use as much memory as it wants; Or,
- Constant memory, in which the system can only maintain a limited constant amount of data per protocol instance.

Due to the large scale of social networks, working with constant memory mode is a more reasonable choice. For example, working in unlimited memory mode, and using for every protocol instance memory in the order of magnitude of the number of participants in the instance, is highly unlikely to scale up. Our protocols use the constant memory mode.

The Protocol Problem. We are interested in one-way, constant memory protocols which atomically transform the social network from one consistent state to another, following a request by a participant to change status. ’Atomic’ means that either all or none of the changes made by the protocol persist in the network when the protocol terminates. We call these protocols Atomic Regret Protocols.

The Concurrency Problem. The problem is how to allow more than one instance of atomic regret protocols for the same event to operate concurrently. These protocol instances must leave the network consistent, while allowing more than one protocol instance to involve the same node.
4.3.4 One-Way Atomic Regret Protocol

Next, we present one-way methods and protocols for atomic regret, with and without concurrency of multiple protocols.

Atomic Regret Protocol. In addition to status values, which are assumed to be consistent, each participant has a stability value, which is one of \{stable, unstable\}, initialized to stable. As in both Facebook and Twitter, whenever a node $v$ changes status or stability value, $v$’s followers are notified automatically (the status is in fact a two-tuple). The participant that wants to change status is the initiator of the protocol instance. The initiator asks to change its status by changing its stability to unstable. Upon the receipt of a stability update, each following participant $u$ decides yes or no without changing $u$’s status, or decides change and changes its stability to unstable - and by this action continues to activate the protocol instance further. Decisions are made based on current statuses and updates received so far. The status of the initiator and participants who decided change are toggled in case of commit, which is an atomic operation that either installs all the changes or leaves statuses unaffected. In case of abort, none are toggled. The system is in charge of commit and abort, as discussed next.

System’s role. In the unlimited memory mode, the system can record which participants are involved in a protocol instance, what their decisions are, and to which participants a commit or abort message has to be sent (and eventually send these messages). As discussed above, this option would potentially require large amounts of memory. In the constant memory mode, which our protocols use, participants have to take an active part in processing commit and abort. In this mode, the system keeps track of the protocol instance progress by maintaining two counters, $c$ and $r$ for the instance; $c$ controls the number of paths in which the instance proceeds; $r$ controls the number of participants whose last decision is no (participants may decide multiple times, as we discuss shortly). We start with $r=0$, $c=1$. In case of a change decision by $v$, $c$ is increased by the number of $v$’s followers minus one. In case of a yes or a no decision, the system decreases $c$ by 1. In addition, in case of a no decision (except for the case where the previous decision was also no), $r$ is increased by 1. If a participant whose last decision is no decides yes or change, $r$ is decreased by 1. When $c$ reaches 0, we say that the protocol instance finished, and the system may issues a
commit (if $r=0$) or an abort (otherwise).

The system could use, e.g., protocol instances ids in order to identify which decision belongs to which instance, since in the constant memory mode it is impossible to record all participants. The initiator, however, may be recorded, and the system can issue a commit or abort message only to the initiator. Followers (of the initiator and of other nodes) will conclude whether the instance commits or aborts, as can be seen in the pseudo-code for this protocol, which appears as Algorithms 19 (initiator) and 20 (participant).

**Multiple decisions by the same node.** Note that it is possible that a node receives more than one message, because of cycles in the social directed or underlying undirected network graph (see Lisa in scenario 1). According to the protocol, the node makes a new decision upon the receipt of each such message. The system considers the last decision (but participants who already decided change once remain unstable). Since a decision may be made only after the receipt of a message, this message necessarily precedes the abort or commit process. Decisions are made assuming that the protocol will eventually commit, (an optimistic approach). If a participant is unstable (i.e., had a change decision) and decides again (say, yes), the eventual commit assumption includes its own change of status.

**Example.** Consider scenario 1 (Figure 4.4). Marge is the initiator, and is therefore unstable after a change decision (system counter $c=2$). Lisa receives this update, decides no and therefore remains stable ($r=1,c=1$). Bart receives the update, decides change and becomes unstable. Homer and Lisa receive this update ($c=2$). Both decide yes ($r=0,c=0$). The system sees that the protocol instance finished without any participant who made a last decision which is no. The system issues a commit message to Marge. Marge toggles status and becomes stable. Lisa and Bart receive the update, and Bart toggles status and becomes stable (Lisa and Homer are already stable). The protocol instance is finished.

### 4.3.4.1 Termination and Effectiveness

**Claim:** The protocol in Algorithms 19 and 20 terminates, and leaves the network consistent.

**Proof.** A participant $p$ activates the protocol when receiving a notification on a status or stability change, or when a commit or abort message is sent from the system.

- Stability change from stable to unstable only happens on a first change decision in an instance,
since according to the algorithm, once a participant becomes unstable, it remains unstable until abort or commit. Therefore, the number of this type of stability changes is finite, since the network is assumed to be finite. Also note that the number of yes/no decisions in the instance is therefore finite, since such decisions can only happen after a change decision of a followed participant.

As discussed in Section 4.3.4, the counter $c$ represents the number of yes or no decisions that have to be made before the system can commit or abort. $c$, initialized to 1, necessarily reaches 0 since $c$ is increased exactly by the number of expected yes or no decisions, after $c$ change decision. Since the number of change decisions is finite, the protocol therefore always commits or aborts.

- Stability changes from unstable to stable happen during a commit or abort process, once for every participant which decided change. The number of stability changes of this type is also finite.
- Status changes happen during a commit process, once for every participant which decided change. The number of status changes is therefore finite.

We conclude that the number of messages in the protocol is finite.

If the protocol aborts, no status changes are installed and the initially consistent network remains consistent. If the protocol commits, all the participant who decided change change status. We argue that these statuses are consistent. A change decision by a participant $p$ means that if a change is installed, $p$ is consistent, given the current statuses of participants that $p$ follows. Every change in these followed participants statuses is either consistent with $p$ (and would trigger a yes decision) or triggers a no decision. In the case of commit, the last decision is necessarily yes. Therefore, $p$ is consistent. The network is therefore consistent when the algorithm terminates. ■
1: wait for stability and status changes from followed participants;
2: if sender is unstable then
3: if v is stable then
4: decide yes, no or change;
5: else
6: //v is not stable (decided change previously)
7: decide yes or no (assuming eventual commit, including v’s own change in status, if v decided change earlier);
8: end if
9: notify system of decision;
10: if decision is change then
11: change stability to unstable;
12: end if
13: end if
14: if sender is stable /*commit or abort*/ then
15: if v is unstable then
16: if sender toggled status then
17: toggle status and change stability to stable;
18: else
19: change stability to stable;
20: end if
21: end if
22: end if
23: end wait for

Algorithm 20: One-Way Atomic Regret - Participant v
4.3.5 One-Way Atomic Regret for Concurrent Instances

We would like to allow multiple protocol instances to operate concurrently, in order to enable multiple participants initiate and participate in protocol instances at the same time. Moreover, we would like to allow multiple instances access the same nodes, and let knowledge gained in one instance be used in another instance (as demonstrated in scenario 3).

The Commit-Abort graph and the Wait-For graph. Protocol instances (related to the same event) use two global structures, the Wait-For Graph (WFG) and the Commit-Abort Graph (CAG). As in transactional systems, the WFG records waiting dependencies between protocol instances (more on transactions in Section 4.7). The CAG is used to enable a higher level of concurrency, where one instance takes advantage of work done by another instance. In the CAG, nodes are protocol instances. Directed edges are of two types, commit edge and abort edge:

- A commit edge, denoted \(((p,q),c)\), where \(p, q\) are nodes, which means that \(p\) may commit if \(q\) commits, and may not commit otherwise.
- An abort edge, denoted \(((p,q),a)\), where \(p, q\) are nodes, which means that \(p\) may commit if \(q\) aborts, and may not commit otherwise.

This is in fact a decision that a participant can make, assuming a prospective abort or commit of another instance. We further discuss the WFG and the CAG later on.

Example. In scenario 3, Section 4.3.2, Jerry decided in \(P_1\) to 'change provided that \(P_2\) commits', and Simba decided in \(P_2\) to 'change provided that \(P_1\) commits'. The corresponding CAG, created by Algorithm 21 (to be discussed), is the graph shown in Figure 4.6 (b). The commit of each of \(P_1, P_2\) depends only on the commit of the other, and therefore the commitment of both preserves consistency.

The Initiator. The initiator runs the same algorithms as in the non-concurrent case.

Multiple protocol instances. We say that a node \(v\) is involved in protocol instance \(p_0\) if \(v\) is the initiator of \(p_0\) or if \(v\) received a message as a consequence of a decision in \(p_0\). Pseudo-code for the algorithm appears as Algorithm 21.

If \(v\) is not yet involved in an instance, and receives a message from a participant, the algorithm
behaves like in the non-concurrent case (Algorithm 20).

If \( v \) is already involved in one or more protocol instances, we denote by \( P=\{p_1,p_2,...,p_n\} \) the set of protocol instances in which \( v \) is involved. We also denote by \( P_y \subseteq P \) the set of instances in which \( v \) decided yes.

The participant running the algorithm (\( v \)) waits for messages from followed participants.

If the sender is stable (Lines 27-28), then the message is a part of a commit or abort process. Algorithm 23 (separated from the main code for the sake of clear presentation) changes \( v \)'s status if necessary.

If the sender is unstable (Lines 8-25), we consider two cases:

- If \( v \) itself is stable, the case is that \( v \) decided only yes or no in each element of \( P \). If \( v \)'s last decision in each of the instances in \( P \) is no, \( v \) can be sure that none of the unstable participants in the instances in \( P \) are to change status (without another decision by \( v \)). Therefore, \( v \) can decide yes, no or change (assuming abort of the instances in which \( v \) decided no, since a commit requires another decision by \( v \)). If \( v \)'s last decision in at least one of the instances in \( P \) is yes (\( P_y \neq \emptyset \)), \( v \) can decide yes or change with commit or abort dependencies on none, some or all the members of \( P_y \) (again, assuming abort of the instances in which \( v \) decided no). Dependencies of different types may be created for different instances. In addition, \( v \) may decide yes, no or change without dependencies.

- If \( v \) is not stable, then there is an instance \( p_c \) in which \( v \) decided change. \( v \) can decide yes with commit or abort dependencies on none, some or all the members of \( P_y \) and \( p_c \), or wait for \( p_c \)'s termination (Line 22). This is useful, for example, if \( v \) wants to decide change provided that \( p_c \) aborts. In this case, \( p_0 \) waits for \( p_c \). When \( p_c \) terminates, \( v \) makes a decision as a stable participant (Line 8), i.e., decides yes, no or change and activates the protocol further. Note that if \( p_c \) commits, \( v \) changes status anyways. Also, as in the case where \( v \) is stable, \( v \) may also decide no.

Then, the CAG and the WFG are updated (see self-explanatory Algorithm 22).
1: **wait for** stability and status changes **from** participants that \( v \) follows;
2: if \( v \) is did not receive a message in any other protocol instance other than \( p_0 \) then
3: switch to Algorithm 20;
4: else
5: let \( P=\{p_1,p_2,\ldots,p_n\} \) be the set of protocol instances in which \( v \) is involved.
   Let \( P_y \subseteq P \) be the of instances in which \( v \) decided yes.
6: end if
7: if sender is *unstable* then
8: if \( v \) is *stable* (i.e., decided only yes or no in each of the elements in \( P \)) then
9: if \( v \) decided only no in all the instances in \( P \) //last decision then
10: decide one of yes, no or change (in \( p_0 \));
11: end if
12: if \( v \) decided yes in at least one instance then
13: decide one of the following decisions:
14: 1. yes with an abort or a commit dependency on one or more members of \( P_y \);
15: 2. change with an abort or a commit dependency on one or more members of \( P_y \);
16: 3. one of no or yes or change (without dependencies);
17: end if
18: else
19: decide one of the following: //\( v \) is unstable
20: let \( p_c \in P \) be the instance in which \( v \) decided change;
21: 4. yes with an abort or a commit dependency on one or more members of \( P_y \cup \{p_c\} \);
22: 5. wait for \( p_c \) to terminate (and then go to Line 8 to process this message as stable);
23: 6. one of no or yes;
24: end if
25: update CAG, WFG and stability (Algorithm 22);
26: else
27: //case: sender (say, in \( p_i \)) is *stable*, need to commit or abort
28: perform Commit or Abort (Algorithm 23);
29: end if
30: end wait for

**Algorithm 21:** One-Way Concurrent Atomic Regret for participant \( v \) in instance \( p_0 \)
1: if decision is 1, 2 or 4 then
2: add to CAG an edge of the form \((p_0, c)\), for each commit dependency;
3: add to CAG an edge of the form \((p_0, a)\), for each abort dependency;
4: end if
5: if decision is 5 //see discussion in Section 4.3.5 then
6: add edge \((p_0, p_c)\) to WFG;
7: end if
8: if decision is change then
9: change stability to unstable; //of \(v\)
10: end if
11: notify system of decision, WFG and CAG changes;

Algorithm 22: Update CAG, WFG and stability

1: if \(v\) is unstable then
2: if sender toggled status (i.e., commit) then
3: toggle status and change stability to stable; //\(v\)’s status changes
4: else
5: change stability to stable;
6: end if
7: end if

Algorithm 23: Commit or Abort

4.3.6 System’s role in Concurrency

As discussed, the system is responsible for sending commit and abort messages to initiators. For concurrency control, the system also monitors and maintains the WFG and the CAG, and decides whether to send commit or abort according to instances. Pseudo-code for the System appears as Algorithm 24. A protocol instance is considered finished as in the non-concurrent case.

A1, A2 and Complexity. A1 is any algorithm which chooses one node (victim) from a cycle in the WFG. The system aborts the corresponding protocol instance, thus resolving the deadlock implied by this cycle. Alternative deadlock resolution could be timeout-based [99], [19]. A2 is
any algorithm which operates on finished protocol instances in the CAG, and decides for each instance, whether it is aborted or committed - without contradicting the dependencies in the CAG. An obvious optimization for A2 is to commit the highest possible number of finished instances.

**Concurrency.** The presented methods can both leverage work done by one instance for the benefit of another (for example and as discussed above, when a waiting instance can rely on a *change* decision of a committing instance), and also when two processes are in synergy (as in scenario 3).

### 4.3.7 The Optimal Abort Problem

The following decision problem is NP-Complete: Given a CAG and an integer $k$, decide whether it is possible to abort at most $k$ finished protocol instances, and commit the rest of the finished instances.

**Proof.** Observe that aborting all the instances never contradicts the dependencies in the CAG. 

1. The problem is in NP, since a Turing machine can guess up to $k$ aborts and verify that no dependency is contradicted.

2. NP-Hardness. By reduction from Vertex Cover problem: given a graph $G$ and a positive integer $k$, does $G$ have a set of vertices of size at most $k$, such that every edge in the graph is incident to a vertex in the set. Vertex Cover is known to be NP-Complete. Let $vc$ be a Vertex Cover instance. We transform $vc$ to an instance of our problem by designating all the edges in $vc$ as abort edges (a linear procedure). The result is $w$, an instance of the optimal abort problem. Observe that for every abort edge $((p,q),a)$, either $p$ or $q$ (or both) must be aborted, in order not to violate the dependency. Therefore, if $vc$ has a vertex cover of size $k$, all edges in $w$ can be covered by the same set, and aborting the transactions corresponding to the nodes in $w$’s cover set satisfies all the abort edges in $w$. If $vc$ has no cover set of size $k$, aborting $k$ transactions in $w$ necessarily leaves an unsatisfied edge in $w$.

### 4.3.7.1 Termination and Effectiveness

**Claim:** The protocol in Algorithms 21 and 24 terminates, and leaves the network consistent.
Algorithm 24: One-Way Concurrent Atomic Regret - System

Proof. Similarly to the non-concurrent case, every instance terminates in either abort or commit after a finite number of messages. The only new case in the concurrent case is the deferred change decision, which results in waiting for another instance. After the waited-for processes are complete, the waiting instance continues as usual. Cycles in the wait-for graph are resolved by algorithm A1.

If a protocol instance aborts, no status changes are installed, and protocol instances which assumed a commit of the protocol instance are also aborted (in order to satisfy the commit edge in the CAG). The initially consistent network therefore remains consistent. If the protocol instance commits, then it is either (1) not dependent on any other instance or (2) all the commit and abort dependencies of this instance are satisfied. In case (1), every installed change is consistent with other changes in the instance. All the participants in the committing instance, whose status has been changed, remain consistent, according to the proof for the non-concurrent case. In case (2), every installed change is consistent with other changes of the instance and with changes of
other instances, on which the instance depends. A2 issues commits and aborts which satisfy these dependencies. Therefore, when all these instances commit or abort, the result is equivalent to the result of committing the instance under discussion after installing all the changes of all the instances on which it has a commit dependency (note that these necessarily commit). Based on the proof for the non-concurrent case, we conclude that the network remains consistent after installing the changes of all the instances. ■

4.4 Two-Way Consistency-Preserving Protocol

We would like to enable protocols to operate efficiently in the two-way communication architecture. Two-way protocols save some of the processing performed by the system, and allow participants not to expose their decisions, which may be private, to the system or as a typically public tweet (but rather to their friends only). As discussed above, the two-way architecture also facilitates the integration of responses by means other than protocol messages into the protocols. In this section, we develop protocols for the two-way architecture and methods for concurrency.

We start with consistency-preserving protocols for one instance.

**Initiator.** Pseudo-code for the algorithm is listed as Algorithm 25. The *initiator* starts a protocol instance by changing its stability value to *unstable*. Initiator’s followers receive an update
on this. The initiator then waits for the time-wise last commit or abort message received from each of its followers, whose algorithm as participants shall be discussed shortly. The time-wise last message is identified using message counters (see below). If all last messages are commit, the initiator starts an atomic commit process by toggling its status and becoming stable. Otherwise, the initiator becomes stable without toggling its status (i.e., an abort process). Note that this is in contrast to the one-way communication architecture, in which the initiator does not decide to commit or abort, but waits for the system’s decision.

Participants. Pseudo-code for the algorithm is listed as Algorithm 26. A participant in a protocol instance, say \( v \), starts participating in the protocol instance when it receives a (stability) update from an initiator or another participant \( u \). \( v \) can make one of three decisions: yes and no, which do not change the participant’s status or stability, or change. change means that in order to preserve consistency, if \( u \) changes status, \( v \) also has to change status. A yes (respectively, no) decision is realized by sending a commit (respectively, abort) message to followed participants that participate in this protocol instance (‘upward’ messages). Participants that receive commit and abort messages from following participants, resend them to instance participants that they follow. Such a message \( m \) is sent only if necessary, i.e., if the last message that \( v \) sent and \( m \) are not both commit or both abort (thus avoiding cycles). A change decision is realized by changing the stability value to unstable (and by this further activating the protocol). In addition (Lines 16 through 23 in Algorithm 26), participants realize an atomic commit or abort process started by the initiator, by changing statuses (in case of commit).

Cycles and Decisions. Similarly to the situation in the one-way architecture, cycles in the directed or undirected social network graph may result in cases where the same protocol instance reaches a participant more than once. According to the protocol, participants make a new decision upon the receipt of a stability change (i.e., a change decision). However, once a participant decides change and by this becomes unstable, subsequent decisions by the same participant may only be yes or no (but the participant remains unstable and would toggle status in case of commit). Each such decision is made based on all the information gathered by the participant so far, and assuming eventual commit of the instance.

Message Counter. The system has a message counter for each protocol instance, which keeps track of the number of protocols messages that are being processed, or are to be processed, by their
1: change stability to unstable
2: wait for last commit/abort message from each follower (until message counter is decreased to 0)
3: if all (last messages) are commit then
4:   toggle status;
5:   change stability to stable;
6: else
7:   change stability to stable; /*abort*/
8: end if
9: end wait for

Algorithm 25: Initiator’s Algorithm

receivers. The counter is incremented before a protocol message is sent. When the processing of a message is complete (including the possible sending of additional messages, with the related counter increments), the counter is decremented. Therefore, when this counter is decremented to zero, no more messages related to the protocol instance will be sent. This information is used in order to determine whether it is necessary to wait for messages, or if all messages in a particular instance have already been received and processed. For the sake of clear and concise presentation, we omit the maintenance of the message counter from the pseudo-code.

4.4.0.2 Termination and Effectiveness

Claim: The protocol in Algorithms 25, 26 terminates, and leaves the network consistent.

Proof. A participant p activates the protocol when receiving a notification on a status or stability change, or when a commit or abort message is sent.

- Stability changes from stable to unstable only happen on a change decision. Since p can decide change at most once per instance, the number of this type of stability changes is finite. Also note that the number of yes/no decisions, which may be taken following a stability change by a followed participant in the instance, is therefore finite.
- Stability changes from unstable to stable happen during a commit or abort process, once for every participant which decided change. The number of stability changes of this type is also finite.
- Status changes happen during a commit process, once for every participant which decided change. The number of status changes is therefore finite.

166
wait for stability and status changes from followed participants and for commit/abort messages from following participants;

if message is from a followed participant then

if sender $u$ is unstable then

if $v$ is stable then

decide yes, no or change;

else

decide yes or no (assuming eventual commit);

end if

if decision is yes (respectively, no) then

send commit (respectively, abort) to followed instance participants (if necessary);

end if

if decision is change then

change stability to unstable;

end if

end if

end if

end wait for

Algorithm 26: Two Way, Single Instance, Participant’s Algorithm ($v$)
A commit/abort message is initially sent by a participant who decided yes/no, respectively. This message can result in at most one message generated by any participant (cycles are prevented). Since the number of yes/no decisions is finite (see above), the number of commit/abort messages is finite. There are no cycles of commit/abort messages, since a participant cannot send two consecutive commit or two consecutive abort messages.

We conclude that the number of messages in the protocol is finite. The protocol therefore stops.

Similarly to the one-way case, if the protocol aborts, no status changes are installed and the initially consistent network remains consistent. If the protocol commits, all the participants who decided change toggle status once. We argue that these statuses are consistent. A change decision by a participant \( p \) means that if a change is installed, \( p \) is consistent, given the current statuses of participants that \( p \) follows. Every change in their statuses is either consistent with \( p \) (e.g., would trigger a yes decision by \( p \) after the stability change) or not consistent, and would trigger a no decision. In the case of commit, the last decision is yes. Therefore, \( p \) is consistent. The network is therefore consistent when the algorithm terminates. ■

### 4.4.1 Two-Way Protocols with Concurrent Instances

As discussed in the context of the one-way case, decisions in social networks are made independently by participants, and it is important to allow multiple protocol instances to operate concurrently. While changes of concurrent instances may well be local, we would also like to allow instances to operate on intersecting parts of the social network, without adversely interfering with each other. Moreover, here too, the methods we present next allow protocol instances (of the same social event) to interact with each other so as to achieve a high level of concurrency. In contrast to the one-way architecture, here participants expose their decisions to friends only, and the system does not have to process a large volume of decisions as in the one-way case. Another advantage of the two-way approach is its ability to naturally integrate responses to protocol messages by means other than protocol messages, such as personal communication, phones and text messages. In this case, the participant receiving the response simulates the receipt of a protocol message. This is not possible in the one-way architecture, in which the system has to receive all decisions directly. We also show an example of how two-way can reuse work by an aborted instance for the benefit of another instance (the formalization of reuse algorithms is the subject of on-going work).
Commit-Abort and Wait-For Graph (CAG,WFG). Here too, we use the CAG and the WFG to record dependencies between protocol instances. In two-way protocols, there are two possibilities for updating the CAG and the WFG. One is to let participants update the graphs directly. The other is to send 'upwards' messages with dependencies, and have the initiator update the graphs. For simplicity, we continue with the former option.

**Initiator’s Algorithm.** To start the protocol, the initiator changes its status from *stable* to *unstable*. As usual, messages to its followers are automatically sent. At this point, the initiator waits for its followers to approve the change. Each follower activates a participant algorithm (to be discussed subsequently). *commit* and *abort* messages have the same meaning as in the non-concurrent case, which is to carry 'upwards' information on participants’ decisions, as well as notify the initiator of a system decision, when sent by the system to the initiator. In addition, the protocol uses abort, commit or wait dependencies between instances, that participants create as a part of their algorithm. If there are no dependencies of the instance on other instances, the initiator decides to commit or abort as in the non-concurrent case. If there are dependencies, the decision whether to commit or abort is made by the system (using the CAG, to be discussed later on). In this case, the initiator waits for the system’s decision (also a *commit* or *abort* message, but from the system), and then triggers the actual commit or abort of the instance by changing its stability value to *stable*. Pseudo-code for this algorithm appears as Algorithm 27.

**Participant’s Algorithm.** The participant running the algorithm, \( v \), waits for instance changing messages, stability and status changes from participants that \( v \) follows (*downward* messages), and for *commit*, *abort* messages from participants that follow \( v \) (*upward* messages). As discussed above, every participant \( p \) directly updates the CAG and WFG dependencies that \( p \) created. Next, we describe the possible responses to these messages. The algorithm for a participant in concurrent mode appears as Algorithm 28.

**Upward Messages.** *commit* and *abort* are processed similarly to their processing in the non-concurrent mode. Algorithm 29 processes upward messages (called in Lines 2-4 of Algorithm 28).

**Downward Messages.** The rest of the algorithm processes downward messages. We consider messages from a stable participant and messages from unstable participant, separated into two al-
1: change stability to unstable
2: wait for last commit/abort message from each follower until message counter is decreased to 0 or for commit/abort from system
3: if any last message from a follower is abort then
4: change stability to stable /*abort*/
5: else if there are no dependencies on other instances then
6: toggle status /*here, all last messages are commit*/;
7: change stability to stable;
8: end if
9: if message is from the system then
10: if message is commit then
11: toggle status;
12: end if
13: change stability to stable /*for both commit and abort*/;
14: end if
15: end wait for

Algorithm 27: Initiator’s Protocol in Concurrent Mode

algorithms for the sake of presentation.

Message from an Unstable Participant. Algorithm 30 treats messages from unstable participants. This case happens when the protocol instance looks for approval from followers (as opposed to the next case, where the protocol executes an aborts or a commit). The algorithm considers two main cases: (1) \( v \) itself is stable, and (2) \( v \) itself is unstable.

In case (1), \( v \) has not decided change in any unterminated instance. If all previous decisions were no, \( v \) is certain about the status of all the participants it follows. Any prospective change will require \( v \)'s approval. \( v \) may therefore freely decide yes, no or change. If \( v \) decided yes in a set of instances, \( v \)'s decision may depend on the outcome of other instances (using dependencies). In case the decision is yes (respectively, no), a commit (respectively, abort) message is sent to instance participants that \( v \) follows.

In case (2), \( v \) has decided change in an instance, say \( p_c \). In this case, another change is not
possible. Instead, \( v \) may defer its response until \( p_c \) terminates. This implies a wait dependency.

**Message from Stable Participant.** Algorithm 31 treats messages from stable participants. This case happens when the protocol instance executes commit or abort, and participants become stable again. If \( v \) is stable, \( v \)'s status is not affected by the commit or the abort in process, and \( v \) does not activate the protocol further. Otherwise, there are two cases: (1) sender toggled status, i.e., the instance is in the process of executing commit and \( v \) changes status; and (2) sender did not toggle status, i.e., the instance is in the process of executing abort and \( v \) does not change status.

**The System and the CAG and WFG.** As in the one-way case, commit, abort, and wait dependencies are updated by participants in the CAG and the WFG. The system is notified by participants which dependencies are created and maintains the graphs. The system is responsible for deciding whether to commit or abort protocol instances that have dependencies on other instances, including identifying situations in which protocol instances may help each other commit (as in the examples above).

The system maintains the message counter associated with each instance, knows when no more messages are expected, in every protocol instance. The system decides whether to commit or abort instances, using algorithms \( A1, A2 \), as in the one-way case (i.e., committing and aborting without violating the dependencies between instances). Note that mutual help situations are reflected as cycles of commit dependencies in the CAG. In this situation, deadlocks are resolved by committing instances, in contrast to isolation-based concurrency, in which deadlocks are always resolved using abort. Deadlocks exhibited in the WFG are resolved as usual [19], [99]. As in the one-way case, terminated (i.e., aborted or committed) instance \( I \) is deleted from the graphs if there are no dependencies of unterminated instances on \( I \).

### 4.4.1.1 Termination and Effectiveness

**Claim:** The protocol in Algorithms 27-28 terminates, and leaves the network consistent.

**Proof** The number of messages in every protocol instance is finite. An instance includes the same messages as in the non-concurrent case. Cycles in the WFG are resolved. The protocol therefore stops.

As for the final state. If the instance aborts, no status changes are installed, and instances
1: wait for stability and status changes from followed participants
   and for commit, abort from followers (until message counter is decreased to 0)
2: if message is from a follower of v (upward messages) then
3: go to Algorithm 29;
4: end if
5: if message is from a followed participant u (downward message) then
6: if sender is unstable then
7: go to Algorithm 30 (unstable sender procedure);
8: else if sender is stable /*commit or abort*/ then
9: go to Algorithm 31 (stable sender procedure);
10: end if
11: end if
12: end wait for

Algorithm 28: Participant’s Algorithm in Concurrent Mode (v)

1: if the last message from all followers is commit then
2: send a commit message to each instance participant u followed by v (unless the last
message sent to the participant is the same);
3: else if at least one of the last messages from all followers is abort then
4: send an abort message to each instance participant u followed by v (unless the last
message sent to the participant is the same);
5: end if

Algorithm 29: Processing by v, of a messages from a follower, in Concurrent Mode

that depend on the instance’s commit will necessarily be aborted (by A2) in order to satisfy the
dependency. The network remains consistent.

If the instance commits, we distinct between two cases, as in the one-way case. If the instance
does not depend on the outcome of other instances, every installed change is consistent with other
changes in the instance, which are also installed. If the instance depends on the commit or abort of
other instance(s), A2’s ensures that all these dependencies are satisfied. Again, as in the one-way
case, the result of the commit is equivalent to installing the changes of the committing instance

172
1. if \( v \) is stable (i.e., not a part of another instance or decided only yes or no in any instance) then
2. if \( v \) is not a part of another instance or decided no in all other instances then
3. decide one of yes or no or change;
4. end if
5. if \( v \) decided yes in at least one instance then
6. decide one of the following:
7. 1. yes with an abort or a commit dependency on one or more members of \( P_y \), or
8. 2. change with an abort or a commit dependency on one or more members of \( P_y \), or
9. 3. one of no or yes or change;
10. send commit/abort upwards if decision is yes or no;
11. update new dependencies in CAG;
12. end if
13. else
14. let \( p_c \in P \) the instance in which \( v \) decided change;
15. decide one of the following:
16. 4. yes with an abort or a commit dependency on one or more members of \( P_y \cup \{p_c\} \), or
17. 5. wait for \( p_c \) to terminate or
18. 6. one of no or yes
19. send commit/abort upwards if decision is yes or no;
20. update new dependencies in CAG, WFG;
21. end if
22. return;

**Algorithm 30:** Processing by \( v \) of a message from unstable sender \( u \)

after committing all the instances on which it has a commit dependency. Therefore, based on the proof for the two-way, non-concurrent case, we conclude that the network remains consistent after commit. ■
1: if \( v \) is stable then
2: 
3: end if
4: if sender toggled status then
5: 
6: else
7: 
8: end if
9: 

**Algorithm 31:** Processing, by \( v \), of a message from stable sender \( u \)

---

**4.4.2 Work Reuse Example**

Next, we give an example of how the two-way communication architecture can be useful for reusing work of aborted instances.

Consider the social network illustrated in Figure 4.8, with participants which decided on their **true** or **false** status w.r.t. an event. The participants consistency requirements are:

- \( D \) comes if both \( A \) and \( B \) come.
- \( C \) comes if \( A \) does not come.

Figure 4.8: Social Network for the example in Section 4.4.2.
\[ G \text{ comes if } C \text{ does not come.} \]
\[ H \text{ comes if } D \text{ does not come.} \]

Suppose that \( A \) and \( B \) wish to independently change status, while ensuring consistency. Each of them starts a protocol instance.

\( A \) starts an instance by notifying \( C \) and \( D \). \( D \) activates the protocol according to its consistency requirements, by involving (directly or indirectly) all the participants that appear with the grey background in Figure 4.8. Eventually, all approve (possibly changing status). \( C \) would like to change status, and notifies \( G \). \( G \) does not approve the change.

Right after \( D \) receives a message in \( A \)’s protocol instance, \( B \) (who wishes to change to \textbf{false}) starts another instance by notifying \( D \) and \( E \). \( E \) approves immediately without changing status. However, let us consider \( D \)’s situation:
- If \( A \)’s instance commits, \( D \) can approve \( B \)’s change to \textbf{false}, since in this case, \( D \)’s status is \textbf{false}.
- If \( A \)’s instance aborts, \( D \)’s status is still \textbf{true}, and in order to approve \( B \)’s change, \( D \) would have to continue activating the protocol instance that \( B \) started.

Therefore, \( B \) waits for \( D \)’s response, which \( D \) can only give when \( A \)’s instance terminates. Since \( G \)’s requirement cannot be met, \( A \) aborts its instance, and sends \( D \) a message to that effect. \( D \) already knows that as a part of the aborted instance, the participants with the grey background reached an agreement about a new consistent state. \( D \) reuses by ’handing over’ the already performed work to \( B \)’s instance by approving \( B \)’s change, without re-performing \( H, K \) and \( I \)’s work.

\( B \) concludes that the status change is possible, and changes status. Direct and indirect followers subsequently change status, if necessary. The protocol instance thus commits without having to re-perform the work on the grey area. This work would have been performed again if protocol instances had been isolated from each other.
4.5 Implementation

4.5.1 Proof of Concept Implementation on Twitter

We implemented a proof-of-concept prototype for one-way protocols, using Twitter’s API, operating through real accounts that we created for this purpose. The API implementation that we use is Winterwell JTwitter [102], called from Java code implementing the protocol logics.

Figure 4.7 is a screen shot of the user representing the system, which ‘sees’ all the tweets. The tweets appear from bottom to top, and are of the form \((status, \text{stability-timestamp})\), where \(status\) and \(stability\) are Boolean. We shall consider Marge, Bart and Lisa of scenario 1 (the rest of the participants in scenario 1 are omitted for the sake of simplicity of presentation). The first three tweets are initialization for the three. The following tweet is Marge’s change to \(unstable\). The protocol then continues, and the next tweet is Bart’s change to \(unstable\) (following his \(change\)) decision. As discussed above, the protocol continues until the system issues a \(commit\) message to Marge. Marge toggles status and becomes \(stable\). Then, Bart toggles status and becomes \(stable\). The protocol instance is concluded.

4.5.2 Simulation of Two-Way Protocols

We also implemented a system simulating the consistency-preserving protocols. The structure of the social network is provided to the system at start up. The user running the simulation defines which participant(s) would like to initiate a consistency-preserving protocol to change their statuses. The user supplies the decisions that participants make as the protocol runs. The system controls the concurrent execution of protocols on intersecting parts of the social network. We demonstrate the on the simple social network illustrated in Figure 4.9.

The log in Figure 4.10 is for a scenario in which Bart would like to change status (to \(false\)), and starts an instance. Lisa and Moe are asked to approve. Moe decides \(yes\), Lisa decides \(change\). Homer and Burns are asked to approve. Both decide \(yes\). The protocol succeeds, commit messages are sent ‘upwards’, and then the commit is materialized ‘downwards’. The old and new statuses of participants also appear in the log.
4.6 Related Theoretical Problems

Next, we analyze the complexity of several problems related to the model and methods discussed so far.

**Consistent Regret.** Consider a social network $N$ in a consistent state $S$, and a participant $v$ that would like to alter its status (i.e., toggle $S(v)$, which is $v$’s status in $S$). The consistent regret problem is to decide whether there exists in $N$ a consistent state $S'$ in which $S(v) \neq S'(v)$. The problem is NP-Complete.

**Proof.** We prove that the Consistent Regret problem is NP-Complete, using a reduction from the NP-Hard join emptiness problem. The join emptiness problem is to decide, given a database instance $D$, whether the natural join of all of $D$’s tables is empty. The problem is NP-Hard [25].

(a) The problem is in NP. A non-deterministic Turing machine can guess a state $S'$ by assigning a status to each participants, and then check that $S'(v) \neq S(v)$ and that $S'$ is consistent by checking
that every consistency table $T_p$ contains a tuple corresponding to the statuses that $S'$ gives to $p$ and to participants that $p$ follows. This procedure is linear.

(b) The problem is NP-Hard. Let $J$ be a Join Emptiness instance. We polynomially construct a consistent regret instance $C$:

- Without loss of generality, we assume that the attributes are binary. If the values in the tables in $J$ are not binary, we transform them to binary as follows. An attribute $A$ with $n$ bit values is transformed to $n$ one-bit attributes $\{A_1, ..., A_n\}$.
- For every such attribute $A_k$ of table $R_i$, we add a node $R_i.A_k$ to $C$.
- For every table $R_i$ in $J$, we create a clique out of the nodes in $C$ which correspond to $R_i$, by adding bidirectional edges between every two of them.
- Every two nodes $R_i.A_k$ and $R_j.A_k$, $i \neq j$ (i.e., attributes with the same name in distinct tables) are bidirectionally connected to each other.
- The consistency table of node $R_i.A_k$ in $C$ is constructed as follows. Let $T_i$ be a table consisting of one attribute for each node in $R_i.A_k$’s clique. The values in $T_i$ are the values of table $R_i$ in $J$. For every node of the form $R_j.A_k$, which is connected to $R_i.A_k$, let $T_j$ be a table with two attributes, corresponding to $R_i.A_k$ and to $R_j.A_k$. The values in $T_j$ are two tuples: $(0,0)$ and $(1,1)$. $R_i.A_k$’s consistency table consists of: (1) the join of $T_i$ with all such $T_j$ (one for every node of the form $R_j.A_k$), and (2) additions to be discussed shortly.
- We add another node to the graph, $v$. All the nodes follow $v$, and every consistency table is added an attribute corresponding to $v$. This attribute’s value is true in all the tuples. In addition, we add another tuple, in which all values are false, to each table.

Figure 4.11: Illustration of the construction in the proof for consistent regret.
We set the status of all the participants to be false (the network is therefore consistent).

This construction is illustrated in Figure 4.11.

The reduction is polynomial, since the number of nodes is linear in the size of the input, the number of edges is square, and creating the tables is also square. If the join of J is not empty, then there is a tuple in which v is true (altered from the original false). The tuple corresponds to the tuple in the non empty join in J. Therefore there exists a consistent state in which v is true. If the join in J is empty, then the only consistent state is the one in which all nodes are false, and therefore v cannot change its status. Thus, the problem is NP-Hard and therefore NP-Complete.

**Multi Regret.** The multi regret problem is to decide, given a consistent network and k participants who would like to change status, whether there exists a consistent state in which the statuses of the k participants are altered.

(a) The problem is in NP. Similarly to the consistent regret problem, a non-deterministic Turing Machine can guess a new state in which \( S(v) \neq S'(v) \) for every participant \( v \) of the \( k \) who wanted to change status, and check its consistency.

(b) The problem is NP-Hard. By reduction from the consistent regret problem, we add \( k-1 \) participants, not connected to any original participant (they are therefore always in consistent state). \( v \) from the consistent regret instance and these \( k-1 \) additional participants are the \( k \) participants, in the multi-regret instance, who would like to change status.

The problem is therefore NP-Complete.

**Contiguous Area Regret.** The contiguous area regret problem is to decide, given a consistent network and a participant \( v \) who would like to change status, whether it is possible to reach a consistent state by altering statuses within a subgraph reachable from \( v \) (intuitively, this means that the new consistent state can be reached by executing one protocol instance).

(a) The problem is in NP. Similarly to the consistent regret problem, a non-deterministic Turing Machine can guess a new state in which \( S(v) \neq S'(v) \), and check that all the participants who
changed status are in reachable from the initiator through participants who also changed status. This check can be done linearly, using, e.g., BFS.

(b) The problem is NP-Hard. By reduction from the consistent regret problem, we add a participant $w$. $w$ follows $v$, and is consistent when its status is as $v$’s status. All the participants in the network follow $w$. Their consistency tables are such that $w$’s status does not affect their decision (a linear addition of rows to the tables). If there is no solution to the original consistent regret instance, the constructed instance does not have a solution as well, since the conditions of the original problem remain. If there is a solution, the same solution solves the constructed instance.

The problem is therefore NP-Complete.

4.7 Related Work

Historically, the evolution and features of social networks have been a subject of research for more than a century. An instructive overview of the subject could be found in [101]. The Internet has brought social networks to the forefront, shaking up traditional assumptions [6]. Recently published research in the area of social networks has been concerned with analysis of social networks structure, privacy and security in social networks, search related issues and more.

Many recently published works in the area of social networks are concerned with analysis of social networks structure, privacy and security in social networks, search related issues and many more. SIGMOD record of March 2008 [9] views the understanding of social networks as a major research challenge. Automation of social networks can be an extension of almost any such existing challenge.

In [81], so-called ‘regret protocols’ operating under the restriction of the one-way communication architecture are presented.

In [82], regret protocols for two-way communication architecture are studied, again with a lower level of concurrency. Theoretical aspects are not considered. Implementation is not provided. Also, the topic of a generic social protocol with modifiers (to initially determine statuses) is not addressed.
In [44], the Semantic Web vision is generalized to encompass e-mail. Models with which users defined goals for e-mail processes are introduced. These do not include a social network. Conceptually, they work on one hop in the social graph.

Concurrency control of uninterpreted transactions (including deadlocks) is extensively covered in [99] and [19]. Unlike transactions, the processes in this paper are interpreted and not isolated. Transactions on trees and DAGs are addressed in [85] and [22], without interaction between nodes. In protocols, the main novelty is based on this interaction. Correctness of transaction processing is based on serializability, whereas here we provide consistency even in cases where the concurrent execution is not equivalent to any serial execution of individual instances. Cooperating Transactions [54] use proclamations of values that may be written to data items in order to ensure serializability.

We are unaware of any work on protocols which run on social networks, with or without concurrency.
Chapter 5

Conclusions

In this thesis, we addressed a variety of novel data management problems, related to three main topics. The first topic is $XPath^L$, a query language which integrates XPath into Datalog, for jointly processing relational and XML data. The second topic is the Query Network model, which is a new Datalog-inspired model motivated by the need to automate interaction social networks. The third topic is protocols for interaction in social networks, which presents an advanced automation method for interaction in social networks.

$XPath^L$ introduces an XPath predicate into the Datalog formalism. The database consists of an XML document (like in XPath) and a collection of tables (like in Datalog). We presented a model for $XPath^L$, and investigated several dialects. We proposed several algorithms for evaluating both recursive $XPath^L$ queries as well as conjunctive queries. Implementation and experimental results for $XPath^L$ were presented. In addition, we characterized the complexity of conjunctive queries on DAGs.

$XPath^L$ is designed as an intermediate language which is not necessarily exposed to users, but allows implementors to include advanced querying features in their languages. $XPath^L$ processes XML and relational data in a loosely coupled fashion. The methods are therefore applicable for other data types, with their predicates, queries and expressions.

We then moved to another novel Datalog querying scenario, motivated by the rise of social networks as the main platform of interaction on the web. In this near-future scenario, social networks participants use automation tools. The Query Network model that we define focuses on automation of proposal and acceptance of connections (friendship) in the social network.
We take a database approach, in which participants define, in a form of a query, whom they would like to propose a connection to, and from whom they would like to accept connection proposals. The union of these queries is a very large Datalog query, in the order of magnitude of the data being queried. This is in sharp contradiction to the traditional database assumption, that queries are small and data are large.

We defined the model and investigated practical and theoretical aspects, as well as implemented the associated algorithms in a system prototype. We experimented with synthetic and real datasets, showing the usefulness of our methods.

The proposal and acceptance of a connection in a social network is in fact a very small protocol that participants in the network use. The next topic of the thesis is automating larger-scale protocols, for more general purposes. This is another step in developing the usability of social networks.

We defined a model for protocols in social networks, and investigated protocols for making initial decisions and for changing existing decisions, to be used by a large number of participants in a social network. Changes are consistency-preserving, which means that new decisions do not violate social requirements that participants in the network defined.

The distributed nature of social networks requires that multiple decision-making processes be carried out concurrently. We develop methods that allow multiple protocol instances to operate concurrently. Not only do the instances not interfere with each other, our methods allow them to operate in synergy and help each other succeed. This increased level of concurrency is possible because we lift the traditional concurrency control isolation requirement and allow protocol instances to interact.

To conclude, the thesis advanced our state of knowledge in very dynamic areas of web technologies, and in particular, social networks.
Chapter 6

Appendix

6.1 Proofs of Expressiveness Results

6.1.1 Definitions

The following definitions will be used in the proofs in Sections 6.1.2-6.1.4. In all of them, \( P \) denotes an \( XPath^L \) program.

A Ground Instance of a Rule. Let \( r \) be a rule in \( P \) and let \( \sigma \) be a substitution for variables in \( r \). If \( r \) has a non-empty body, then the ground instance of a rule \( r \) with respect to \( \sigma \) is a tree of height one. The tree’s root (or the instance’s head) is a ground instance of \( r \)’s head predicate, in which variables are substituted by \( \sigma \). The tree’s leaves are ground instances of \( r \)’s subgoals in which variables are also substituted by \( \sigma \). If \( r \) has an empty body, then the instance is the instance’s head only. If \( \sigma \) satisfies (respectively, does not satisfy) all the body subgoals, we say that the rule instance is valid (respectively, invalid).

Stratum of a Predicate. We define the stratum of a predicate in \( P \). Given a predicate \( p \) in \( P \), we say that:

- \( p \) is in stratum \( 0 \) if \( p \) corresponds to an EDB relation.

- \( p \) is in stratum \( k \) (where \( k > 0 \)) if
There exists a rule $r$ in $P$ whose head is $p$ and whose body contains a distinguished subgoal whose predicate is in stratum $k - 1$, where a distinguished subgoal is either a negated subgoal or an EDB fact; And

There does not exist a rule $r$ whose head is $p$ and whose body contains a distinguished subgoal in stratum greater than $k - 1$.

Partial Execution Tree.

A partial execution tree in stratum $k$ (where $k > 0$) is a finite tree in which every intermediate node, say $v$, and its children, say $u_1, u_2, ..., u_m$ satisfy the following conditions:

- $v$ and its children are a valid ground instance of a rule in $P$.
- If there exists $u_i$ which is a leaf in the tree, then $u$ is either:
  - A fact whose predicate is defined in a stratum $l$ where $l < k$. If $u_i$ is not negated, then a partial execution tree whose root is $u_i$ exists (in stratum $l$). If $u_i$ is negated, then a partial execution tree whose root is $u_i$ does not exist.
  - A ground instance of a rule without a body in $P$.

A partial execution tree in stratum 0 is simply an EDB fact.

Stratified Execution Tree. Let $n$ be the maximal stratum number for a predicate in $P$. A stratified execution tree for $P$ is an ordered set containing $n + 1$ elements. The $i$-th element ($i = 0, 1, 2, ..., n$) is the set of all partial execution trees in stratum $i$. It is called the $i$’th execution stratum.

From this definition it follows that for stratified programs, if $g(t_1, ..., t_k)$ is a ground subgoal whose corresponding predicate is defined in stratum $k$, then

- A negated ground subgoal $\neg g(t_1, ..., t_k)$ is satisfied iff stratum $k$ in the stratified execution tree does not contain a partial execution tree whose root is the ground subgoal $g(t_1, ..., t_k)$.
- A non-negated ground subgoal $g(t_1, ..., t_k)$ is satisfied iff stratum $k$ in the stratified execution tree does contain a partial execution tree whose head is the ground subgoal $g(t_1, ..., t_k)$.

Sufficient Execution Tree. Given a satisfied ground subgoal $g$ whose corresponding predicate is defined in stratum $m$ of a stratified execution tree $Q$, a sufficient (stratified) execution tree for $g$ in
$Q$ is an ordered set containing $m + 1$ elements. The $i$'th element ($i = 0, 1, 2, ..., m$) is a subset of execution stratum $i$ in $Q$. It is called the $i$'th sufficient execution stratum. The elements in the sufficient execution strata are as follows.

- Sufficient execution stratum $m$ contains exactly one partial execution tree whose head is $g$.
- Sufficient execution stratum 0 is identical to execution stratum 0 in the stratified execution tree.
- Sufficient execution stratum $j$ ($0 < j < m$) is a subset of execution stratum $j$ in $Q$. For every non-negated leaf $l$ in a tree in sufficient execution strata $j + 1, j + 2, ..., m$ whose predicate is defined in stratum $j$, sufficient execution stratum $j$ contains a tree whose head is $l$ (note that at least one such exists, since $g$ is satisfied). For every negated leaf $l$, sufficient execution stratum $j$ contains all the trees that are required to determine that the negated ground subgoal $l$ can not be satisfied, i.e., all the possible valid trees whose root corresponds to $l$. Note that in this case, the number of elements in the sufficient execution stratum may not be finite.

**Notation.** An execution tree (partial, sufficient or stratified) which describes the execution of a program $P$ on a database instance $D$ is denoted $T_{P(D)}$.

**Return Value.** The return value of a program is true iff there exists, in the stratified execution tree, a rule instance whose head is the target predicate.

**Remarks.** (1) The same satisfied ground subgoal may have one or more sufficient execution trees. (2) If a rule $r$ includes a negated subgoal $\neg q$ in its body, and if the predicate $q$ is in stratum $k$ in $P$, then the predicate in $r$’s head is a predicate which is defined in stratum greater than or equal to $k + 1$. (3) A program has a true return value iff there exists a sufficient execution tree for the target predicate.

**Equivalence of Programs** For the sake of readability, we repeat the definition of equivalence of Boolean programs. Boolean programs $P_1 = <r_1, t_1>$ and $P_2 = <r_2, t_2>$ are equivalent if for all database states, after evaluation, the relation for IDB relation $t_1$ is empty iff the relation for IDB
relation $t_2$ is empty.

**Example.** Consider the following program and database instance:

1. $sg(X,X) \leftarrow \text{descendant::*[Y,X], root(Y)}$
2. $sg(X,Y) \leftarrow sg(Z_1,Z_2), \text{child::b[Z_1,X], child::b[Z_2,Y]}$
3. $res() \leftarrow sg(X,Y), a(X,Y)$

$res$ is the target predicate.

The EDB instance is:

The relations $\text{Root, Val and XPath}$ - as defined above.

$A = \{(id_2, id_4)\}$ (The nodes marked by $\ast$ and $\ast\ast$).

The XML document is:

```xml
<?xmlversion = "1.0" encoding = "ISO-8859-1" ?>
<b> /id is id_1
...... <b> /id is id_2 $\ast$
............ <b/> </b> /id is id_3
...... </b>
...... <b/> </b> /id is id_4 $\ast\ast$
</b>
```

The document root element, which cannot be seen here, has id $id_0$.

Denote the length of the path from the root to a node $n$ by $l(n)$. The program returns $\text{true}$ iff a tuple in the relation $a$ consists of ids of nodes that are of the "same generation". Namely, if the two nodes form together an element in the set $\{<x,y> | x \text{ and } y \text{ are nodes and } l(x)=l(y)\}$.

The valid ground instances of the rules (or simply instances of rules) in the program and the database instance follow.

- For the first rule:
  
  \[ sg(id_1, id_1) \leftarrow \text{descendant::*(id_0, id_1), root(id_0)} \]
  \[ sg(id_2, id_2) \leftarrow \text{descendant::*(id_0, id_2), root(id_0)} \]
  \[ sg(id_3, id_3) \leftarrow \text{descendant::*(id_0, id_3), root(id_0)} \]
  \[ sg(id_4, id_4) \leftarrow \text{descendant::*(id_0, id_4), root(id_0)} \]
• For the second rule:
\[ sg(id_2, id_4) \leftarrow sg(id_1, id_1), child::b(id_1, id_2), child::b(id_1, id_4) \]

• For the third rule:
\[ res() \leftarrow sg(id_2, id_4), a(id_2, id_4) \]

The rule instances above include all the possible instances. They are all parts of execution trees which belong to execution stratum 1. EDB facts like root(id_0) and descendant::*(id_0, id_3) are in stratum 0. In order to describe the successful proof of the target predicate, not all of the instances in the stratified execution tree are necessary. For example, three instances in stratum 1 which appear in the partial execution tree illustrated in Figure 6.1 are sufficient to prove that Res contains the empty tuple after evaluation. A sufficient execution tree for res is illustrated in Figure 6.2.

Example. The next example is for a program with negation, which has a true return value. The program is:

1. \( bad(X) \leftarrow child::b[X, Y] \)
2. \( res() \leftarrow parent::b[X, Y], \neg bad(X) \)

The XML tree is the same tree as in the previous example.

An element in bad is an element which has a "b"-labeled child element. The predicate bad is in stratum 1, and the predicate res - in stratum 2. In order to satisfy \( \neg bad(X) \), X has to be substituted
Figure 6.2: A sufficient execution tree with two strata

Figure 6.3: Stratified execution tree for a program with stratified negation.
by an element \( x_0 \), s.t. \( bad(x_0) \) is not the root of a tree in stratum 1. We show that this program has a \textbf{true} return value. In Figure 6.3, the stratified execution tree for this program is illustrated. EDB facts are in execution stratum 0. In execution stratum 1, all the instances for the first rule are presented. It is the only rule with predicate \( bad \) in its head, and \( bad \) is the only predicate in stratum 1. Substitutions in which \( X \) is substituted by \( id_0 \), \( id_1 \) or \( id_2 \) do not satisfy the second rule in the program, since there exists an execution tree for \( bad(id_0) \), \( bad(id_1) \) and for \( bad(id_2) \). According to the above definition for execution trees, substitutions in which \( X \) is substituted by \( id_0 \), \( id_1 \) or \( id_2 \) do not satisfy the rule since they make the predicate \( \neg bad(X) \) evaluate to \textbf{false}. However, every other substitution for \( X \), namely \( id_3 \) or \( id_4 \) does satisfy \( \neg bad(X) \), as there is no execution tree for \( bad(id_3) \) or \( bad(id_4) \) in execution stratum 1. As can be seen in Figure 6.3, the second rule (whose head is the target predicate) has valid instances, and therefore the return value of the program is \textbf{true}.

6.1.2 \( L \equiv L_v \)

\textbf{Theorem.} Every Boolean program in \( L \) has an equivalent program in \( L_v \) and vice versa.

\textbf{Proof.} \( L_v \subseteq L \). Since an \( L_v \) program is also an \( L \) program, every \( L_v \) program is its own equivalent \( L \) program. Therefore, \( L_v \subseteq L \).

\( L \subseteq L_v \). We present \( V \), a transformation from \( L \) to \( L_v \). \( V \) takes a Boolean program in \( L \), \( P \), and transforms it to \( V(P) \), an equivalent program in \( L_v \). \( V \) starts with the rules of \( P \) and consists of the following steps:

- A rule \( bad(E) \leftarrow xpath(W, E, X), xpath(W, E, Y), \neg self ::*(X, Y), root(W) \) is added to \( P \). Intuitively, an XPath expression is "bad" if its resulting node set has more than one member. We call an expression whose result set is a singleton a \textit{logical pointer}. For every node \( n \), there always exists an expression \( e_{\text{pointer}} \) w.r.t. the database XML tree, which navigates from the root only to \( n \), and therefore an expression equivalent to \( e_{\text{pointer}} \) is in \( E_d \) (defines in Section (2.2.4)) where \( d \) is the document whose root is bound to \( W \).

- Subgoal variables can either be bound to a value object \( id \) (type \( v \) arguments) or a non-value
object id (type $o$ arguments). For every rule $r$, for every one of the $2^n$ possibilities to assign a type $o$ or $v$ to each of the $n$ variables in the rule’s head and IDB subgoals, a new rule $r'$ is added. In $r'$, the head and IDB subgoals predicate names are extended with a string over \{o, v\} which corresponds to the specific assignment of types to the head and subgoals’ variables in the order in which they appear in the predicate (adornment is consistent among multiple occurrences of the same variable). For every variable $X$ in the head (respectively, $Y$) with adornment $o$, i.e., associated with an $o$ argument position, (respectively, $v$), the predicate $\neg val(X, X)$ (respectively, $val(Y, Y)$) is added to the body. For constants, only the adornment $v$ is considered, and no predicates are added to the body. Note that the adornment specifies which are non-value and which are value objects among the arguments, as the added $val(\cdot, \cdot)$ predicates impose the corresponding bindings. The original rule, $r$, is deleted.

- We introduce another adornment type, $l$ (logical pointer). A logical pointer is an expression whose result set when evaluated at the root is a singleton. We define the following macro expansion. Whenever we write in a program $lPointer(X, E_X)$, we actually mean to write the following three subgoals $root(R), xpath(R, E_X, X), \neg bad(E_X)$, where $R$ is a new, unused variable. In a rule, every variable $X$ adorned by $o$ is replaced by a variable $E_X$ with adornment $l$ (in both body and head), and the subgoal $lPointer(X, E_X)$ is added to the body. This is done in order to translate from a logical pointer ($E_X$) to its corresponding object id (bound to $X$), so that references to $X$ in the rest of the body retain their original semantics.

For example, $V$ applied to the third rule in the ”same generation” program, i.e., to:

1 $sg(X, X) \leftarrow descendant::*[Y,X], root(Y)$
2 $sg(X, Y) \leftarrow sg(Z_1, Z_2), child::b[Z_1,X], child::b[Z_2,Y]$  
3 $res() \leftarrow sg[X,Y], a(X,Y)$

would produce the following program:

Added rule:

$$bad(E) \leftarrow xpath(W,E,X), xpath(W,E,Y), \neg self::*[X,Y], root(W)$$

For the first rule:

$$sgll(E_X, E_X) \leftarrow descendant::*[Y,X], root(Y), \neg val(X,X), \neg val(X,X), lPointer(X, E_X)$$
Note that only the first rule can be instantiated. However, all the rules are shown for the sake of completeness of the example.

Second rule:

\[ sg_{ll}(E_X, E_Y) \leftarrow sg_{ll}(E_{Z_1}, E_{Z_2}), \text{child}::b[Z_1, X], \text{child}::b[Z_2, Y], \neg \text{val}(X, X), \neg \text{val}(Y, Y), \text{lPointer}(X, E_X), \text{lPointer}(Y, E_Y), \text{lPointer}(Z_1, E_{Z_1}), \text{lPointer}(Z_2, E_{Z_2}) \]

\[ sg_{lv}(X, Y) \leftarrow sg_{lv}(E_{Z_1}, Z_2), \text{child}::b[Z_1, X], \text{child}::b[Z_2, Y], \text{val}(X, X), \text{val}(Y, Y), \text{lPointer}(Z_1, E_{Z_1}) \]
We argue that for every program $P$ in $L$, for every database state $D$, the programs $V(P)$ (i.e., the result of applying $V$ on $P$) and $P$ have the same Boolean value.

**Proposition.** Consider the stratified execution tree $T_{P(D)}$. If there is a sufficient execution tree
$T_{suf}$ in $T_{P(D)}$ for the ground subgoal $p(b_1, ..., b_m)$, which corresponds to an IDB predicate $p$, then there is a corresponding sufficient execution tree $T'_{suf}$ in $T_{V(P)(D)}$ for the ground (IDB) subgoal $p_{a_1...a_m}(c_1, ..., c_m)$, where for every $i$ (1 ≤ $i$ ≤ $m$):

1. If $b_i$ is the id of a value object, then $a_i = v$ and $c_i = b_i$.

2. If $b_i$ is the id of a non-value object, then $a_i = l$ and $c_i$ is the logical pointer navigating from the root to $b_i$ (or, by abuse of language, the macro notation $lPointer(b_i, c_i)$ "holds").

Also, if there is a sufficient execution tree $T'_{suf}$ in $T_{V(P)(D)}$ for the ground IDB subgoal $p_{a_1...a_m}(c_1, ..., c_m)$, then there is a corresponding execution sufficient tree $T_{suf}$ in $T_{P(D)}$ for the ground subgoal $p(b_1, ..., b_m)$, where for every $i$ (1 ≤ $i$ ≤ $m$).

3. If $a_i = v$ then $c_i$ is the id of a value object, and $b_i = c_i$.

4. If $a_i = l$ then $c_i$ is the logical pointer navigating from the root to $b_i$ (or, by abuse of language, the macro notation $lPointer(b_i, c_i)$ "holds").

**Proof.** We prove the proposition by induction on $n$, the ordinal number of (sufficient) execution strata in $T_{suf}$.

**Overall Induction Basis.** $n = 0$. No execution stratum except for stratum 0 exists in $T_{suf}$ or in $T'_{suf}$. According to the definition of a sufficient tree, stratum 0 contains the EDB facts only. The proposition, which is about IDB predicates, vacuously holds, since there are none in stratum 0.

**Overall Induction Hypothesis.** For every ground subgoal $p(b_1, ..., b_m)$ which appears in stratum $n - 1$ at most in $T_{suf}$, there exists a corresponding sufficient execution tree $T'_{suf}$ in $T_{V(P)(D)}$ for the ground IDB subgoal $p_{a_1...a_m}(c_1, ..., c_m)$ where $a_i, b_i$ and $c_i$ satisfy conditions 1 and 2 in the proposition. Also, for every ground subgoal $p_{a_1...a_m}(c_1, ..., c_m)$ in stratum $n - 1$ at most in $T'_{suf}$, there exists a corresponding sufficient execution tree $T_{suf}$ in $T_{P(D)}$ where $a_i, b_i$ and $c_i$ satisfy conditions 3 and 4 in the proposition.

**Overall Induction Step.** In the proof of the overall induction step, we call the proof that $T_{suf}$ implies $T'_{suf}$ "direction 1", and call the proof that $T'_{suf}$ implies $T_{suf}$ "direction 2".
Consider $T_n$, the partial execution tree in stratum $n$ ($n > 0$) in $T_{suf}$ whose head is $p(b_1, ..., b_k)$ (only one exists). By induction on $k$, $T_n$’s height, we prove that a corresponding partial execution tree $T'_n$ whose root is $p_{a_1...a_m}(c_1, ..., c_m)$ exists (and therefore, a sufficient execution tree $T'_{suf}$ for the ground subgoal $p_{a_1...a_m}(c_1, ..., c_m)$ exists in $T_{V(P)(D)}$). Also, if such a $T'_n$ exists, then a corresponding $T_n$ exists (and therefore, a sufficient execution tree $T_{suf}$ for $p(b_1, ..., b_k)$ exists in $T_{P(D)}$).

$T_{q,p}$ is the notation for a partial execution tree in stratum $q$ whose height is $p$.

**Inner Induction Basis.** $k = 0$.

- **Direction 1.** $T_{n,0}$ is a node. $T_{n,0}$ must be an IDB fact that is the instance of the head of a rule $r$ in $P$ which has no body. $r$ is therefore:

  $$p(b_1, b_2, ..., b_m) \leftarrow .$$

  Rules are safe, therefore in $r$, there are zero or more constants in the head and no variables. Constants are value objects only, therefore, the set $V(r)$, the set of rules produced by $V$ from $r$ contains only one rule, with $v$ adornments only. Constants are not changed by $V$, therefore:

  $$V(r) = \{p_{vv...v}(b_1, b_2, ...b_m) \leftarrow .\}$$

  The instance of this rule is the ground subgoal $p_{a_1...a_m}(c_1, ..., c_m)$ where for all $i$, $a_i = v$ and $c_i = b_i$. This instance satisfies condition 1 (and, vacuously, 2) in the proposition. We therefore propose this instance as $T'_{n,0}$, and conclude the proof of direction 1 in the inner induction basis.

- **Direction 2.** $T'_{n,0}$ is a node. $T'_{n,0}$ must be an IDB fact that is the instance of the head of a rule $r'$ in $V(P)$ which has no body. $r'$ is therefore of the form:

  $$p_{a_1...a_m}(c_1, ..., c_m) \leftarrow .$$

  And the rule $r$ from which $r'$ was produced by $V$ is:

  $$p(b_1, b_2, ..., b_m) \leftarrow .$$

196
Rules are safe, therefore in \( r \), there are zero or more constants in the head and no variables. Constants are value objects only, therefore for all \( i \), \( a_i = v \) and \( c_i = b_i \). The instance of this rule, which is the ground subgoal \( p(b_1, b_2, ..., b_m) \) satisfies condition 3 (and, vacuously, 4) in the proposition. We therefore propose this instance as \( T_{n,0} \) and conclude the proof of direction 2 in the inner induction basis.

**Inner Induction Hypothesis.** We assume that if \( T_{n,k-1} \), a partial execution tree in execution stratum \( n \) \( (n > 0) \) of height \( k - 1 \) at most exists in \( T_{P(D)} \), then a corresponding \( T'_n \) exists in \( T_{V(P)(D)} \). Also, if \( T_{n,k-1} \) exists, then \( T_{n,k-1} \) exists.

**Inner Induction Step.**

- **Direction 1.** Consider \( T_{n,k} \), a partial execution tree of height \( k \) in stratum \( n \) in \( T_{P(D)} \), and assume its root is \( p(b_1, b_2, ..., b_m) \). This root, together with its children is an instance of a rule in \( P \), say \( r \), and is denoted \( r_{\text{instance}} \). Let \( V(r) \) be the set of rules in \( P' \) that was produced from \( r \) by \( V \). Consider the rule \( q \) in \( V(r) \); \( q \) is the one rule that was created from \( r \) based on the particular typing of arguments in \( r \) which corresponds to the actual binding to value and non-value object ids in \( r_{\text{instance}} \). Next, based on \( r_{\text{instance}} \) we propose an instance \( q_{\text{instance}} \) for \( q \) by substituting its variables, and prove that this is a valid instance of \( q \). Consider a subgoal \( g \) in \( r \) and the corresponding ground subgoal \( g_{\text{ground}} \) in \( r_{\text{instance}} \). \( g_{\text{ground}} \) has a sufficient execution tree in \( T_{P(D)} \). If \( g_{\text{ground}} \) is an EDB fact, then it remains one after applying \( V \) on \( P \), as the EDB was not changed and EDB predicates in the program are not changed by the transformation. If \( g_{\text{ground}} \) is an IDB fact, then according to the inner induction hypothesis, a corresponding sufficient execution tree for \( g'_{\text{ground}} \), the corresponding (adorned) ground subgoal in \( q \), exists in \( T_{V(P)(D)} \). Every IDB subgoal in \( r \) has a corresponding subgoal in \( q \). However, there are subgoals in \( q \) which do not have a non-adorned corresponding subgoal in \( r \). These subgoals are the subgoals added to \( q \) by \( V \) (as opposed to the subgoals discussed so far, that exist originally in \( r \) and may only be adorned by \( V \)). We call those subgoals in \( q \) which are an adorned version of IDB subgoals in \( r \) ”modified subgoals”, and call the subgoals added to \( q \) by \( V \)”added subgoals”. There are two types of added subgoals:

1. Subgoals represented by the macro \( lPointer(X, E_X) \), namely
   
   \[ \text{root}(R), \text{xpath}(R, E_X, X), \neg \text{bad}(E_X). \]
2. Subgoals of the form $\text{val}(X, X)$ or $\neg\text{val}(X, X)$.

We continue building $q_{\text{instance}}$ by proposing the following substitution for the variables in the above types of added subgoals (respectively):

1. According to $V$, a variable $X$ in the third position of the an $\text{xpath}$ predicate also appears in at least one of the modified subgoals. Let the variable which appears with $X$ in the same ground $\text{xpath}$ subgoal be called $E_X$. The ground subgoals for the modified subgoals discussed above represent a binding for $X$, say $x_0$. In the added subgoals, we substitute $X$ by $x_0$ and $E_X$ by the logical pointer for $x_0$ (which must exists). $R$ is substituted by the object id of the document’s root. This proposed substitution satisfies the three subgoals $\text{root}(R), \text{xpath}(R, E_X, X), \neg\text{bad}(E_X)$.

2. According to the hypothesis, arguments of ground subgoals conform to their adornment. $V$ is defined so that $\text{val}(X, X)$ (respectively, $\neg\text{val}(X, X)$) is added if the adornment for $X$ is $v$ (respectively, $o$). Therefore, a binding $x_0$ for $X$ necessarily satisfies $\text{val}(X, X)$ if $X$’s adornment is $v$, or $\neg\text{val}(X, X)$, otherwise.

**Remark.** Note that since $r_{\text{instance}}$ is a valid instance of a rule, it follows that in case where a variable has multiple occurrences in the modified subgoals of $q$, they are all substituted by the same value in $q_{\text{instance}}$. Otherwise conditions 1 and 2 in the proposition would not have hold for subgoals in $r_{\text{instance}}$.

By this we proposed a ground subgoal for every subgoal in the body of $q$. For the variables of $q$’s head, which are a subset of the variables in $q$’s body, we propose the same bindings as in the body. By this we completed proposing $q_{\text{instance}}$. In this proposal, all the body subgoals are satisfied, and substitutions of multiple occurrences of the same variable are kept consistent. Therefore, $q_{\text{instance}}$ is a valid instance of $q$. Also note that substitutions conform to their adornment in $q_{\text{instance}}$’s body, and therefore they also do so in the head.

We now consider the head of $r_{\text{instance}} (p(b_1, b_2, \ldots, b_m))$ and the head of $q_{\text{instance}} (p_{a_1} \ldots a_m (c_1, \ldots, c_m))$ and show that they satisfy conditions 1 and 2 in the proposition.

1. If $a_i = v$, then $b_i$ is a value object id and in the proposal for $q_{\text{instance}}$ remains unchanged, or $c_i = b_i$. 

198
2. If \( a_i = l \), then \( c_i \) is a substitution of a variable \( E_X \). As shown above, \( c_i \) is chosen to be the logical pointer to \( b_i \).

This concludes the proof of direction 1. We proposed a rule instance \( q_{\text{instance}} \) whose head’s adornments correspond to the bindings in \( r_{\text{instance}} \) and showed that it is valid. Then, we showed that it meets conditions 1 and 2 in the proposition.

- **Direction 2.** Consider \( T'_{n,k} \), a partial execution tree of height \( k \) in stratum \( n \) in \( T_{V(P)(V)} \), and assume its root is \( p_{a_1...a_m}(c_1, c_2, ..., c_m) \). This root, together with its children is an instance of a rule in \( V(P) \), say \( q \), and is denoted \( q_{\text{instance}} \). Also, denote by \( r \) the rule in \( P \) from which \( q \) was produced by \( V \). Based on \( q_{\text{instance}} \), we propose an instance \( r_{\text{instance}} \) for \( r \) by substituting its variables, and prove that this is a valid instance of \( r \). Consider a subgoal \( g' \) in \( q \) and the corresponding ground subgoal \( g'_{\text{ground}} \) in \( q_{\text{instance}} \). \( g'_{\text{ground}} \) has a sufficient execution tree in \( T_{P(D)} \). If \( g'_{\text{ground}} \) is an EDB fact, then it remains one in \( T_{P(D)} \). If \( g'_{\text{ground}} \) is an IDB fact, then according to the inner induction hypothesis, a corresponding sufficient execution tree for \( g_{\text{ground}} \), the corresponding (not adorned) ground subgoal in \( r \), exists in \( T_{P(D)} \).

**Remark.** Note that since \( q_{\text{instance}} \) is a valid instance of a rule, it follows that in case where a variable has multiple occurrences in subgoals of \( r \), they are all substituted by the same value in \( r_{\text{instance}} \). Otherwise, conditions 3 and 4 in the proposition would not have hold for the subgoals in \( q_{\text{instance}} \)’s body.

By this we proposed a ground subgoal for every subgoal in the body of \( r \). For the variables of \( r \)’s head, which are a subset of the variables in \( r \)’s body, we propose the same bindings as in the body. By this we completed proposing \( r_{\text{instance}} \). In this proposal, all the body subgoals are satisfied, and substitutions of multiple occurrences of the same variable are kept consistent. Therefore, \( r_{\text{instance}} \) is a valid instance of \( r \).

We now consider the head of \( r_{\text{instance}} \) \( p(b_1, b_2, ..., b_m) \) and the head of \( q_{\text{instance}} \) \( p_{a_1...a_m}(c_1, ..., c_m) \) and show that they satisfy conditions 3 and 4 in the proposition:

3. If \( a_i = v \), then \( b_i \) is a value object id and in the proposal for \( r_{\text{instance}} \) remains unchanged, or \( c_i = b_i \).

4. If \( a_i = l \), then \( c_i \) is a substitution of a variable \( E_X \). As shown above, \( c_i \) is chosen to be the logical pointer to \( b_i \).
This concludes the proof of direction 2 of the inner induction step. We proposed a rule instance \( r_{\text{instance}} \) in which variables are bound to values which conform to the corresponding adornments in \( q_{\text{instance}} \), and showed that the instance is valid. Then, we showed that it satisfies conditions 1 and 2 in the proposition.

This concludes the proof of the overall induction step.

We now return to the theorem and prove it using the proposition proved above.

Assume \( P(D) = \text{true} \). A sufficient execution tree in \( T_{P(D)} \) (for the target predicate of \( P \)) exists. According to the proposition, a corresponding sufficient execution tree (for the target predicate of \( V(P) \)) in \( T_{V(P)(D)} \) exists too. Thus, \( V(P)(D) = \text{true} \).

Assume \( V(P)(D) = \text{true} \). A sufficient execution tree in \( T_{V(P)(D)} \) (for the target predicate of \( v(P) \)) exists. According to the proposition, a corresponding sufficient execution tree (for the target predicate of \( P \)) in \( T_{P(D)} \) exists too. Thus, \( P(D) = \text{true} \). In conclusion, \( P(D) = \text{true} \iff V(P)(D) = \text{true} \), which concludes the proof of the theorem.

**Remark:** The trivial proof that \( P' \) is in \( L_v \), i.e., meets the syntactic requirements of \( L_v \), is omitted.

### 6.1.3 \( L_c \preceq L \)

There exists a Boolean \( L \) program that can not be expressed in \( L_c \).

**Proposition.** For every \( k \in \mathbb{N} \), there exists an XML document \( d \) with no text nodes and a Boolean matrix \( A \) s.t. \( e_A \) is in \( E_d \) (and therefore \( \neg \text{Bad}(e_A) \) holds) and \( |e_A| \geq k \), where \( |e| \) is the number of bits used in the representation of \( e \) (Intuitively, the (simple) proposition is that for every possible length of expression, there exists an ”allowed” expression which is longer, w.r.t. a certain document).

**Proof.** Assume to the contrary that there exists \( K \in \mathbb{N} \) s.t. for every \( d \) and for every \( A \), \( |e_A| < K \). There exist at most \( 2^K \) such expressions. Consider a document with \( N = 2^K \) elements. There exists an expression which navigates from the root to the \( i^{\text{th}} \) element and only to it. Denote the matrix corresponding to such an expression by \( A_i \). Therefore, there are \( N \) expressions in \( E_d \). Now, consider a document with \( 2N \) elements. The number of expressions which navigate to one node only from the root is \( 2 \cdot 2^K \). Therefore, there is an expression \( e_{\text{long}} \) (in fact, many of them) s.t.
\[|e_{long}| > K, \text{ a contradiction, which concludes the proof.}\]

Let us define \( P \), the following program in \( L \):

\[
\text{res()} \leftarrow u(E_1), \// * (W, X), \text{xpath}(X, E_1, Y), \text{root}(W).
\]

\( P \) returns \textbf{true} if there is in relation \( U(\cdot) \) (which corresponds to the predicate \( u(\cdot) \)) an expression which navigates between two nodes in document \( q \), whose tree’s root is in \( \text{root}(\cdot) \), and is in \( E_q \), where \( q \) denotes the XML document. We argue that an equivalent program in \( L_c \) does not exist.

Assume to the contrary that \( P_c \), an \( L_c \) program equivalent to \( P \), does exist.

According to the proposition, there exists a document \( d \) with no text nodes and a matrix \( A \) corresponding to an expression \( e_A \) s.t. \( |e_A| > |P_c| \). \( e_A \) can therefore not appear as a constant in the \( P_c \). As a consequence, every occurrence of \( e_A \) in a corresponding execution tree is a result of a binding to a variable rather than a constant. \( P \) returns \textbf{true} when evaluated on a database whose EDB contains \( d \)’s root in \( \text{root}(\cdot) \) and the relation \( u(\cdot) \) contains the tuple \( (e_A) \). Therefore, \( P_c \), which is equivalent to \( P \), returns \textbf{true} as well, and there exists a sufficient execution tree \( T \) for \( \text{res} \) built of \( P_c \)’s rules. We argue that \( T' \), which is created from \( T \) by replacing all the occurrences of \( e_A \) by \( \text{self}\::*\text{/}e_A \), is a sufficient execution tree for \( P_c \)’s target predicate in the case where the EDB is modified so that \( \text{self}\::*\text{/}e_A \) is in \( u(\cdot) \) and \( e_A \) is not (referred to as the \textit{modified EDB} below). This would be a contradiction, since \( \text{self}\::*\text{/}e_A \) not in \( E_d \) (with respect to any document) and nonetheless, \( P_c \) returns \textbf{true} when \( u(\cdot) \) contains this expression. We prove this by induction on \( n \), the number of execution strata in \( T \). Note that for a non-empty program, \( n > 0 \) since any instance of a rule is in execution stratum 1 at least. Therefore, the induction basis is \( n = 1 \).

**Overall Induction Basis.** \( n = 1 \). \( T \) has two strata, stratum 0 and stratum 1. Since \( T \) is a sufficient tree, there is exactly one partial execution tree \( T_1 \) in execution stratum 1. Execution stratum 1 in \( T' \) therefore also contains only one partial execution tree, \( T'_1 \). We prove that \( T'_1 \) is a partial execution tree for \( P_c \)’s target predicate, by induction on \( k \), the height of \( T_1 \) (and \( T'_1 \)). By this we prove that \( T' \) is a stratified execution tree for \( P_c \)’s target predicate. Note that \( e_A \) and \( \text{self}\::*\text{/}e_A \) can not appear as a constant in \( P_c \), since each one of the two is longer than the whole program.

- **Inner Induction Basis.** \( k = 0 \). \( T_1 \) is a node. Therefore, \( T'_1 \) is a node. This implies that the node is an instance of a rule \( r \) without a body. Since rules are safe, there are no atoms in the
body. The only possibility for such a node to be changed by the procedure which modifies \( T \) to create \( T' \) is a case where \( e_A \) appears as a constant in \( r \). However this is not possible since \( e_A \) is longer than the whole program. In conclusion, if \( k = 0 \) then \( T_1 \) and \( T'_1 \) are identical.

- **Inner Induction Hypothesis.** If \( Q \) is a partial execution tree in stratum 1 of height \( k - 1 \) at most, then \( Q' \) (i.e., \( Q \) in which \( e_A \) was replaced by \( self::*/e_A \)) is a partial execution tree w.r.t. the modified EDB. Also, if \( Q' \) is a partial execution tree, then \( Q \) is a partial execution tree.

- **Inner Induction Step.** Let \( T_1 \) be a partial execution tree of height \( k \). Consider the rule instance \( r_1 \) whose head is the root of \( T_1 \). The nodes in the body of \( r_1 \) are either:

  - Non-negated subgoals. Each subgoal corresponds either to a predicate in stratum 1 or to a predicate in stratum 0.

    * If a subgoal \( s \) corresponds to a predicate in stratum 1, then after changing \( T \) to \( T' \), are each the root of a valid execution subtree, according to the induction hypothesis.

    * If a subgoal \( s \) is an EDB subgoal (i.e., a leaf \( T_1 \) which corresponds to a stratum 0 predicate), there are two cases. If \( s \) is not \( u \), then \( s \) remains unchanged and is a valid ground subgoal since the only EDB relation that was changed is \( u \). If \( s \) is \( u \), then it is possible that a node \( u(e_A) \) is changed to \( u(self::*/e_A) \). The result is a valid ground subgoal, since \( self::*/e_A \) was added to \( u \).

  - Negated subgoals. Here too, each subgoal corresponds either to a predicate in stratum 1 or to a predicate in stratum 0.

    * Consider a negated ground subgoal in \( T_1 \) which corresponds to a stratum 1 predicate, \( \neg g \), and denote by \( \neg g' \) the corresponding ground subgoal in \( T'_1 \). We argue that if \( \neg g \) is a satisfied negated ground subgoal in \( T_1 \) then \( \neg g' \) is satisfied in \( T'_1 \). Assume to the contrary that \( \neg g' \) is unsatisfied in \( T'_1 \) (or, \( g' \) is satisfied). According to the hypothesis, \( g \) is satisfied. This implies that \( \neg g \) is not satisfied, a contradiction.

    * If a negated subgoal \( \neg s \) is an EDB subgoal, we consider two cases. If \( s \) is not \( u \), then \( s \) remains unchanged and is a valid ground subgoal since the only EDB
relation that was changed is \( u \). If \( s \) is \( u \), we argue that \( s \) was not changed. The only possibility for a change to occur is if a node \( \neg u(e_A) \) is changed to \( \neg u(self::/e_A) \). However, \( \neg u(e_A) \) can not appear in \( T_1 \) since it evaluates to \textit{false}.

In conclusion, all \( r_1 \) subgoals are satisfied.

We argue that \( r_1 \) is valid, and therefore \( T_1' \) is a partial execution tree. For this to hold, substitutions of the same variable have to be consistent. By creating \( T_1' \), we change every occurrence of \( e_A \) by \( self::*/e_A \), preserving consistency among multiple occurrences of the same variable that might exist (in the head or body). This consistency and the validity of the ground subgoals in \( r_1 \)'s body prove that \( r_1 \) is valid and that \( T_1' \) is a partial execution tree.

Given \( T_1' \) in stratum 1, it can be shown, using the exact same considerations used in this induction, that the replacement of all the occurrences of \( self::*/e_A \) by \( e_A \) results a partial execution tree \( T_1 \).

**Overall Induction Hypothesis.** If \( T_{n-1} \) is a partial execution tree in stratum \( n-1 \), then \( T_{n-1}' \) is a partial execution tree w.r.t. the modified EDB. If \( T_{n-1}' \) is a partial execution tree, then \( T_{n-1} \) is also a partial execution tree.

**Overall Induction Step.** Consider a partial execution tree \( T_n \) in stratum \( n \). The proof that \( T_n' \) is valid too is done by induction on \( k \), the height of \( T_n \) (and \( T_n' \)), is identical to the inner induction in the overall induction basis, as it considers a partial execution tree in a certain stratum whose possibly negated leaves have, or do not have, partial execution trees in lower strata. This concludes the overall induction.

\( self::*/e_A \) does not have the structure of expressions in \( E_d \). Therefore \( P_c \) is not equivalent to \( P \), since \( P_c \) returns \textit{true} when \( u(\cdot) \) contains the non-\( E_d \) expression \( self::*/e_A \). To complete the proof of the theorem, we point out that every \( L_c \) program is its own \( L \) equivalent. In conclusion, \( L_c \prec L \).
6.1.4 $L^+ \equiv L^+_c$

Every Boolean $L^+$ program applied to an EDB instance that does not include XPath expressions in relations or as values of text nodes in the tree has an equivalent $L^+_c$ program, and vice versa.

**Proof.** (1) $L^+_c \subseteq L^+$. Since an $L^+_c$ program is also an $L^+$ program, every $L^+_c$ program is its own equivalent $L^+$ program. Therefore, $L^+_c \subseteq L^+$.

(2) $L^+ \subseteq L^+_c$. We present a transformation $V$ from $L^+$ to $L^+_c$, and argue that for every program $P \in L^+$, $V(P)$ and $P$ are equivalent. We define the set $Z$ to be the set of expressions which explicitly appear in the program and, in addition, containing the expression `/descendant-or-self::*`. In addition, we call a variable in a rule is an *e-variable* if at least one of its occurrences is as the second argument of an XPath predicate.

For a given program $P$, $V(P)$ is the following program:

- For each rule $r \in P$ and for each possible substitution $\sigma$ which substitutes e-variables only with elements from $Z$, $V(P)$ includes a rule $r_\sigma$ in which:
  - Every e-variable in a predicate other than `xpath` is replaced by the (constant) expression assigned to it by $\sigma$.
  - Every `xpath` predicate with an e-variable is replaced by a predicate corresponding to an expression relation. The relation is the one whose name coincides with the (constant) expression assigned to the e-variable by $\sigma$. The relation’s parameters are the first and second parameters of the original `xpath` predicate, in their original order. For example, consider the predicate `xpath(X, E, Y)`. If $\sigma$ substitutes $E$ by $exp_0$, then $r_\sigma$’s body includes $exp_0(X, Y)$ and does not include `xpath(X, E, Y)`.

- The original rule $r$ is removed.

- Rules with no e-variables are included and remain unchanged.

**Example.** If $Z = \{e_1, e_2, /descendant-or-self::*\}$ and a rule $r_0$ has two e-variables, the transformed program includes $3^2$ rules instead of $r$, one for every possible substitution of e-variables with elements from $Z$. Given $P \in L^+$, then $V(P) \in L^+_c$, since negation was not introduced and
e-variables were replaced by constants.

Let $D$ be an EDB instance containing no expressions. We prove that $V(P)(D) = \text{true}$ iff $P(D) = \text{true}$.

Assume $V(P)(D) = \text{true}$. Therefore, there exists a sufficient execution tree $T$ for the target predicate, in which stratum 1 contains a partial execution tree whose head is $V(P)$’s target predicate. Note that the absence of negation implies that every execution tree has two strata. Stratum 0 (for EDB facts) and stratum 1 (for rule instances). We argue that $T'$, a sufficient execution tree which proves $P$’s target predicate exists. Consider $t$, an instance in $T$ of a rule $r$ ($r$ is in $V(P)$). Let $\rho$ be the substitution corresponding to $t$, and let $r'$ be the rule in $P$ from which $r$ was created. If we extend $\rho$ to substitute e-variables in $r'$ by the elements from $Z$ which appear in $t$ (possibly as expression relations names), we get a satisfying substitution for the rule $r'$. This is true for every rule instance in $T$. Thus, $P(D) = \text{true}$.

Assume that $P(D) = \text{true}$. Therefore, there exists a sufficient execution tree in $T_{P(D)}$ for $P$’s target predicate. For those rules whose instances are in the tree, e-variables can appear (1) only in XPath predicates or (2) in both XPath and IDB predicates. By assumption, EDB relations do not contain expressions, therefore if predicates corresponding to EDB predicates have e-variables, the rule in which they appear can not appear in the tree because they can never be satisfied. An e-variable can not appear in the second position of a $\text{val}(:,\cdot)$ predicate where the first argument is a text node object, since in this case too, the XPath predicate which caused the variable to be an e-variable is evaluated to $\text{false}$ as there are no valid XPath expressions in text nodes of the tree. In case (1) above, an e-variable $E$ appears only in XPath predicates (and not in the head), say in the predicate $\text{xpath}(X,E,Y)$. Every substitution which satisfies $\text{xpath}(X,E,Y)$ also satisfies the expression relation predicate $\text{/descendant-or-self::*}(X,Y)$. We can therefore replace all the occurrences of every such $\text{xpath}$ predicate by the expression relation, and produce an equivalent program $P'$ that will be discussed from this point. Note that every e-variable which also appears in an IDB relation in $P$ does so in $P'$ too.

We argue that expressions in IDB relations (case (2) above) can only be expressions which appear explicitly in the program. Assume to the contrary that there exists an IDB relation $r$ in which there is an expression $e$ such that $e$ does not appear explicitly in the program. Then, there exists
in $T$ a rule instance whose head is a ground subgoal corresponding to the relation $r$ that one of its parameters is $e$, i.e., $r(..., e, ...)$. We prove that such a subgoal can not exist in $T$. The contradiction implies that only expressions which appear explicitly in the program can be in an IDB relation.

It is clear that stratum 0 does not contain $r(..., e, ...)$, since stratum 0 is defined to contain only EDB fact and $r$ is an IDB relation.

We prove by induction on $k$, the height of $T_1$, which is the partial execution tree in execution stratum 1 in $T$ (and whose head is the ground target predicate), that $r(..., e, ...)$ does not appear in $T_1$.

- **Induction Basis.** $k=0$. If $T_1$ is of height zero, then the partial execution tree is simply a node. This implies that the node is an instance of a rule without a body. Since rules are safe, there are no variables in the body. The only possibility for $e$ to be inserted to $r$ by such a rule is therefore to explicitly appear in the program.

- **Induction Hypothesis.** If a partial execution tree in stratum 1 of height $k - 1$ or less contains an expression $e$ (as an argument of a ground subgoal), then $e$ explicitly appears in the program.

- **Induction step:** Consider a partial execution tree $p$ in stratum 1 of height $k$, and consider the rule instance whose head is the root of the $p$. The root’s children are each the root of a subtree whose height is $k - 1$ at most. Since rules are safe, expressions can be inserted to the IDB only if they are bound to an e-variable in the body. In other words, if an expression $e$ appears in the head then it also appears in the body. According to the induction hypothesis, if an expression $e$ appears in the body, it also appear explicitly in the program.

In conclusion, there are no expressions in the EDB and only those expressions which explicitly appear in the program can be in the IDB. This means that e-variables are limited to be bound to expressions which appear in the program\(^2\).

Thus, in every case where a sufficient execution tree $T'$ for the target predicate of $P'$ (in $T'_{P'(D)}$) represents a binding of an expression, it is necessarily an expression from the set $Z$. Every instance of a rule $r$ in $T'$ therefore corresponds to a rule $r_\sigma$ in $V(P')$. In $r_\sigma$, each e-variables $E$, is replaced by a constant equal to the binding of $E$ represented by the tree or by an expression relation. A

\(^2\)Remark: Expressions in the program may include the expression / descendant-or-self::* that may appear in $P'$. 

206
proposed sufficient execution tree $T_v$ which proves $V(P')(D) = \text{true}$ is a tree where every instance of a rule $r$ in $T'$ is replaced by the corresponding instance of a rule $r_\sigma$ in which (the remaining) variables are substituted by the same values as in $T'$. The existence of the sufficient execution tree implies $V(P')(D) = \text{true}$ and therefore $V(P)(D) = \text{true}$ which concludes the proof.

6.2 Query Networks Datasets Information

Figure 6.4 lists information about the datasets used in the various Query Networks experiments.
Bibliography


[86] SAXON, the XSLT and XQuery Processor. http://saxon.sourceforge.net/

[87] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/


[107] XQuery 1.0. An XML Query Language. 2007 http://www.w3.org/TR/xquery/

מודלים ורשויות לאפליקציות אינטגר條件
מקדימות לאטרומציה בﴩתו חברית
רועי רון
מודלים ושיטות לאפליקציות אינטגרט
מתקדמים לאוטומציה ברשויות בניית

תייבר על מתקד

לשם מילוי תلجنة של הדרישה לעבלת התואר
דוקטור לפילוסופיה

ורעי רון

הנהל להנחות ונטינות – מכון טכנולוגי לישראל
חתםsteam בודז
חרפת ינואר 2012
המחבר נגעしゃ באזורייה פורפ רודר שמריאלי

הפכולת למדעי המחשב

אני מודה לטכניון ולקרן ג

אריג מרדת לשכנון לקוף נייעובס על התוכנה הנדיבת בחרטומזי

הפקולטה למדעי המחשב
הנושאים האמורים או עיכוד של מדע הנחון - של מיחזור לזריזנות באהת הנושאים. אנשי מיחזור יוצרים מתהליך המהווה של מדע הנחון האתחול פעולת תרבות או תרבות של מדע הנחון - של מיחזור לזריזנות באהת הנושאים.

שלב זה מתدفاع על ידי הנוסח פָּרְדִיקְטָב פָּרְדִיקְטָב (_android פָּרְדִיקְטָב ל-לְפָרְדִיקְטָב). אנשי התחזק את השפהшенזורה XPathL (_android פָּרְדִיקְטָב ל-לְפָּרְדִיקְטָב). פָּרְדִיקְטָב יוצרים פָּרְדִיקְטָב של מדע הנחון האתחול פעולת תרבות או תרבות של מדע הנחון - של מיחזור לזריזנות באהת הנושאים.

לפיכך, במ İnsan של קירוב של מדע הנחון האתחול פעולת תרבות או תרבות של מדע הנחון - של מיחזור לזריזנות באהת הנושאים, פָּרְדִיקְטָב פָּרְדִיקְטָב (android פָּרְדִיקְטָב ל-לְפָּרְדִיקְטָב). אנשי התחזק את השפהщенזורה XPathL (android פָּרְדִיקְטָב ל-לְפָּרְדִיקְטָב). פָּרְדִיקְטָב יוצרים פָּרְדִיקְטָב של מדע הנחון האתחול פעולת תרבות או תרבות של מדע הנחון - של מיחזור לזריזנות באהת הנושאים.

לפיכך, במ İnsan של קירוב של מדע הנחון האתחול פעולת תרבות או תרבות של מדע הנחון - של מיחזור לזריזנות באהת הנושאים, פָּרְדִיקְטָב פָּרְדִיקְטָב (android פָּרְדִיקְטָב ל-לְפָּרְדִיקְטָב). אנשי התחזק את השפתהשהון XPathL (android פָּרְדִיקְטָב ל-לְפָּרְדִיקְטָב). פָּרְדִיקְטָב יוצרים פָּרְדִיקְטָב של מדע הנחון האתחול פעולת תרבות או תרבות של מדע הנחון - של מיחזור לזריזנות באהת הנושאים.

לפיכך, במ İnsan של קירוב של מדע הנחון האתחול פעולת תרבות או תרבות של מדע הנחון - של מיחזור לזריזנות באהת הנושאים, פָּרְדִיקְטָב פָּרְדִיקְטָב (android פָּרְדִיקְטָב ל-לְפָּרְדִיקְטָב). אנשי התחזק את השפתהשהון XPathL (android פָּרְדִיקְטָב ל-לְפָּרְדִיקְטָב). פָּרְדִיקְטָב יוצרים פָּרְדִיקְטָב של מדע הנחון האתחול פעולת תרבות או תרבות של מדע הנחון - של מיחזור לזריזנות באהת הנושאים.
The model uses a formalism "Datalog" to refer to "data talk." Two is a new model of the subject of DataLog in which participants use automated tools to describe social networks. Motivation for such automated tools is the increased utility of data talk for the participants. They work in models of networks that gradually become business tools for everything. The social networks in which a user is interested are defined in Datalog. This is in contrast to the traditional model of data talk. Data using various optimization methods, which are not effective in relation to the size of data. News and information are different in the first model than in the second model, in which the decision problem, for example, is studied. We study problems related to the models, such as whether a network is fully justified, for example, a network of requests that will lead to the creation of a fixed number of new links in the network. We show that there are problems that are easy to solve in the model without conditions. Therefore, it is not difficult to solve problems that are solved in the model with conditions. This is the subject of the third issue. It is the purpose of protocol meetings that participants in the social network can coordinate their actions and carry out tasks that require cooperation between a large number of participants who are not necessarily directly related to each other. Technion - Computer Science Department - Ph.D. Thesis PHD-2012-14 - 2012
הסוג הראשון הוא פרוטוקולים המתאימים קביעה ראשונית של סטטוס ביחס לארוע ופרמטרים שונים המעצבים את סוגי הודעות בפרוטוקולים אלה יש דוגמאות לפרמטרים贿כהויה. התנהגות הפרוטוקול הלאוי ודרישות לתשובות וא magna עקביות של הודעות בשטח זה, והאהמה של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, אנו מתאימים לשינויים של הודעות בשטח זה, והאהמה של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והאהמה של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות רבות את הפרוטוקולים. הפרוטוקולים מתאימים לשינויים של הודעות בשטח זה, והfeeds של הפרוטוקולים זה אנו ממחישים באמצעות דוגמאות gratuites de soql בקישור זה.
לפי כן, התזה זו תורמת לחקר קיימים מספר מודלים ובעיות תдерשות הקשורים
בתחเหมาะสมות האתחנות של האינטרניט ולפי הש蒌 וב(מתודות בופת
והברית), ומיצגים אלגוריתמים, מעמוד ומקורות תאורטיים.