

Distributed data management with the rule-based language: Webdamlog

Émilien Antoine

▶ **To cite this version:**

Émilien Antoine. Distributed data management with the rule-based language: Webdamlog. Databases [cs.DB]. Université Paris Sud - Paris XI, 2013. English. NNT : . tel-00908155v5

HAL Id: tel-00908155

https://theses.hal.science/tel-00908155v5

Submitted on 20 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

Université Paris-Sud

École Doctorale d'informatique de Paris-Sud

Laboratoire Spécification et Vérification (LSV) – ENS de Cachan

Discipline : Informatique

Thèse de doctorat

Soutenue le 5 décembre 2013 par

Émilien Antoine

Gestion des données distribuées avec le langage de règles: *Webdamlog*

Distributed data management with the rule-based language: *Webdamlog*

Directeur de thèse :	Serge Abiteboul	D.R. Inria Saclay
Composition du jury :		
Présidente du jury :	Nicole Bidoit	Prof. Univ. Paris-Sud
Rapporteurs :	Christine Collet	Prof. Grenoble INP
	Pascal Molli	Prof. Univ. Nantes
Examinateurs :	Bogdan Cautis	Prof. Univ. Paris-Sud
	David Gross-Amblard	Prof. Univ. Rennes 1

Résumé

Notre but est de permettre à un utilisateur du Web d'organiser la gestion de ses données distribuées *en place*, c'est à dire sans l'obliger à centraliser ses données chez un unique hôte. Par conséquent, notre système diffère de Facebook et des autres systèmes centralisés, et propose une alternative permettant aux utilisateurs de lancer leurs *propres pairs* sur leurs machines gérant localement leurs données personnelles et collaborant éventuellement avec des services Web externes.

Dans ma thèse, je présente *Webdamlog*, un langage dérivé de datalog pour la gestion de données et de connaissances distribuées. Le langage étend datalog de plusieurs manières, principalement avec une nouvelle propriété *la délégation*, autorisant les pairs à échanger non seulement des faits (les données) mais aussi des règles (la connaissance). J'ai ensuite mené une étude utilisateur pour démontrer l'utilisation du langage. Enfin je décris le moteur d'évaluation de *Webdamlog* qui étend un moteur d'évaluation de datalog distribué nommé *Bud*, en ajoutant le support de la délégation et d'autres innovations telles que la possibilité d'avoir des variables pour les noms de pairs et des relations. J'aborde de nouvelles techniques d'optimisation, notamment basées sur la provenance des faits et des règles. Je présente des expérimentations qui démontrent que le coût du support des nouvelles propriétés de *Webdamlog* reste raisonnable même pour de gros volumes de données. Finalement, je présente l'implémentation d'un pair *Webdamlog* qui fournit l'environnement pour le moteur. En particulier, certains adaptateurs permettant aux pairs *Webdamlog* d'échanger des données avec d'autres pairs sur Internet. Pour illustrer l'utilisation de ces pairs, j'ai implémenté une application de partage de photos dans un réseau social en *Webdamlog*.

Mots clefs Distribution ; Datalog ; Base de connaissances ; Pair à pair ; Gestion de données du Web.

Abstract

Our goal is to enable a Web user to easily specify distributed data management tasks *in place*, i.e. without centralizing the data to a single provider. Our system is therefore not a replacement for Facebook, or any centralized system, but an alternative that allows users to launch their *own peers* on their machines processing their own local personal data, and possibly collaborating with Web services.

We introduce *Webdamlog*, a datalog-style language for managing distributed data and knowledge. The language extends datalog in a number of ways, notably with a novel feature, namely *delegation*, allowing peers to exchange not only facts but also rules. We present a user study that demonstrates the usability of the language. We describe a *Webdamlog* engine that extends a distributed datalog engine, namely *Bud*, with the support of delegation and of a number of other novelties of *Webdamlog* such as the possibility to have variables denoting peers or relations. We mention novel optimization techniques, notably one based on the provenance of facts and rules. We exhibit experiments that demonstrate that the rich features of *Webdamlog* can be supported at reasonable cost and that the engine scales to large volumes of data. Finally, we discuss the implementation of a *Webdamlog* peer system that provides an environment for the engine. In particular, a peer supports wrappers to exchange *Webdamlog* data with non-*Webdamlog* peers. We illustrate these peers by presenting a picture management application that we used for demonstration purposes.

Keywords Distribution ; Datalog ; Knowledge Base ; Peer to Peer ; Web Data Management.

Contents

Acknowledgement **ix**

Résumé en Français **xi**

1 Introduction **1**

2 State of the Art **3**
2.1 Distributed Information Systems 3
2.1.1 Distributed systems 3
2.1.2 Distributed databases 4
2.1.3 Data on the Web 4
2.1.4 Peer-to-peer systems 5
2.1.5 Social networks 6
2.1.6 Contribution 7
2.2 Knowledge bases 7
2.2.1 Processing knowledge 7
2.2.2 Datalog 8
2.2.3 Distributed datalog 10
2.2.4 Provenance and optimization 11
2.2.5 Contribution 11
2.3 Webdam exchange 12

3 *Webdamlog* language **15**
3.1 Model of data 17
3.1.1 Informal presentation 17
3.1.2 Formal definitions 19
3.2 Key observations 24
3.3 Expressive power 27
3.3.1 Traces and simulations 28
3.3.2 Expressivity results 28
3.4 Convergence of *Webdamlog* 33

3.4.1 Positive *Webdamlog* . . . 33
3.4.2 Strongly-stratified *Webdamlog* . . . 41

4 *Webdamlog* rule engine 53
4.1 Datalog inside . . . 53
4.2 Connection between *Bud* and *Webdamlog* . . . 54
4.2.1 *Webdamlog* computation on *Bud* . . . 54
4.2.2 Implementing Webdamlog rules . . . 56
4.3 Optimization of the evaluation . . . 60
4.4 Optimization for view maintenance . . . 62
4.4.1 Provenance graphs . . . 62
4.4.2 Deletions . . . 65
4.4.3 Running the fixpoint . . . 65
4.5 Performance evaluation . . . 66
4.5.1 Cost of delegation . . . 67
4.5.2 Cost of dynamism . . . 69

5 Architecture of a *Webdamlog* peer 77
5.1 Peer architecture . . . 78
5.1.1 Event-driven system . . . 80
5.1.2 Module interactions . . . 81
5.2 Wrappers . . . 83
5.3 Demonstration . . . 86
5.3.1 Wepic application . . . 86
5.3.2 Demonstration Scenario . . . 90
5.3.3 Access control . . . 93

6 User Study 95
6.1 *Webdamlog* tutorial . . . 95
6.2 Test . . . 98
6.3 Results . . . 101

7 Conclusion 103

Self references 105

External references 107

Acknowledgement

The very first person I would like to thank in these acknowledgments is my advisor Serge Abiteboul. He leads me all along the long and winding road that are the three years of a Ph.D. thesis. I claim that he is the best advisor that a student could have, although he is the only Ph.D. advisor I have ever had so my judgment may be biased. I have always been amazed to see him implied in so many activities in scientific matters in the most prestigious places as well as in other domains as writing and sculpture. In spite of his busy schedule, I always had the opportunity to talk about my work when needed and even have enjoyable gabbing at lunch. To summarize, I will quote what Val Tannen once told me about Serge Abiteboul "In the neighborhood of Serge, we are all the cool people of databases" and I totally agree that is why thanks to him I meet so many interesting people I wish to thank. First I wish to thank Julia Stoyanovich with whom I have worked and met in many occasions during conferences or summer schools. Then I am grateful to all the Webdam members I had the opportunity to meet. Especially Alban Galland with whom I started to collaborate as a Ph.D. student and Pierre Bourhis who are the two previous student of Serge; and also Jules Testard who worked with me during its internship in Webdam. Once again thanks to all the people in Webdam, it would be too long to make list them here so check yourself on the Webdam website the list of participants. Moreover it has been a pleasure to work with all the members of the LSV, the laboratory in which I spent most of my time, but also the member of the current OAK team formerly LEO in which I started my first year as a Ph.D. student. Eventually, let me acknowledge the member of my jury: Christine Collet and Pascal Molli for the relevant remarks and reviews of my thesis; Nicole Bidoit, Bogdan Cautis and David Gross-Amblard who accepted to be my examiners. Thanks again to David Gross-Amblard who have been my internship tutor before my thesis. He allows me to meet Philippe Rigaux who is just like Serge another very cool people of databases.

In addition to the people who collaborate with me, I would like to thank co-workers. More precisely, the people who shared an office with me: Wojciech

Kazana, Nadime Françis and Marie Van Den Bogaard, thanks for the nice time and enjoyable discussions about nearly everything that could be turned into a riddle. The other people of the fourth floor of the LSV, especially Thomas Chatain for the enjoyable coffee breaks, Sandie Balaguer for the gardening breaks on the balcony and Luc Segoufin and Cristina Sirangelo for gathering people at noon and pleasant discussion at lunch.

To conclude, this thesis would not have been possible without my family and above all my parents who had always been involved in several exciting activities and who shared their interest with their children. From ornithology, gardening, carpentry to jazz, science and politic all these matters always kept my curiosity aroused. Also thanks to my brother who has traveled through all the Americas and post the story of his journey that entertained me. Finally I am grateful to many friends who helped me to escape from my Ph.D. thesis from times to times: Sainte-Tempérance, Gaudy Whynot, Grolodie and all the "Lorrains" and "Normands" I had the pleasure to meet in Paris ; and the good old friends before my arrival in Paris Sab, Flo, Ninie, Émilie, Raph, JB.

Résumé en Français

Le volume d'informations présentes sur la toile[1] s'accroît exponentiellement. Les utilisateurs comme les compagnies partagent de plus en plus leurs données, qui se trouvent distribuées sur les nombreuses machines qu'ils possèdent, ou via des services Web où ils stockent leurs informations sur des machines externes. En particulier, l'émergence de l'infonuagique[2] et des réseaux sociaux permet aux utilisateurs de partager encore plus de données personnelles. La multitude de services spécialisés offrant chacun une expérience différente à l'utilisateur complique énormément la gestion de ces informations et la collaboration des ces services dépasse rapidement l'expertise humaine. Les informations manipulées par les utilisateurs ont de nombreuses facettes : elles concernent des données personnelles (photos, films, musique, mails), des données sociales (annotations, recommandation, liens sociaux), la localisation des données (marque-pages), les informations de contrôle d'accès (mots de passe, clés privées), les services Web (moteur de recherche, archives), la sémantique (ontologies), la croyance et la provenance. Les tâches exécutées par les utilisateurs sont très variées : recherches par mots clefs, requêtes structurées, mise à jour, authentification, fouille de données et extraction de connaissances. Dans cette thèse, nous montrons que toute cette information devrait être modélisée comme un problème de gestion d'une base de connaissance distribuée. Nous soutenons aussi que datalog et ses extensions forment une base formelle sûre pour représenter ces informations et ces tâches. Ce travail fait partie du projet ERC *Webdam* [ERC13] sur les fondations de la gestion des données de la toile. Le but de ce projet est de participer au développement de fondations formelles unifiées pour la gestion de données distribuées, le manque actuel de telles fondations ralentissant les progrès dans ce domaine.

Pour illustrer la problématique nous pouvons considérer *Joe* un utilisateur typique de la toile. *Joe* possède un blog hébergé sur Wordpress.com sur lequel il poste des critiques de films qu'il a visionné récemment. *Joe* possède aussi un compte Facebook et Gmail pour communiquer avec ses amis, ainsi

[1] Web

[2] cloud computing

qu'un compte Dropbox pour stocker une partie des données qu'il souhaite partager. *Joe* aimerait automatiser une tâche qu'il effectue régulièrement manuellement. Chaque fois qu'il poste une nouvelle critique sur son blog *Joe* souhaiterait informer ses amis qu'un nouvel article est disponible et mettre à leur disposition le fichier du film qu'il vient de regarder. Cette tâche est possible à automatiser pour un programmeur en écrivant un script adhoc. Cependant *Joe* n'étant pas programmeur il est obligé de s'authentifier sur Wordpress.com, Facebook, Gmail et Dropbox pour y poster sa critique, envoyer un message à tous ses amis et envoyer le fichier du film. Nous proposons dans cette thèse un système permettant à *Joe* de continuer à utiliser les services Web qu'il affectionne tout en spécifiant à son ordinateur des tâches qu'il pourrait accomplir automatiquement.

Contributions

Les contributions de cette thèse sont les suivantes :

- Je présente *Webdamlog*, un nouveau langage à base de règles pour la gestion de données distribuées qui combine dans un cadre formel les règles déductives de datalog avec négation pour la définition des faits intentionnels et les règles actives de datalog $\neg\neg$ pour les mises à jour et les communications. Le modèle met un accent fort sur la dynamique et les interactions typiques du Web 2.0, principalement grâce à une nouveauté du langage *Webdamlog*, la *délégation* de règles permettant aux pairs de collaborer. Ce modèle est à la fois suffisamment puissant pour spécifier des systèmes distribués complexes et suffisamment simple pour permettre une étude formelle de la distribution, de la concurrence et de l'expressivité dans un système de pairs autonomes.

- Je présente l'implémentation du moteur d'évaluation de programmes *Webdamlog* qui étend le moteur datalog distribué avec mise à jour nommé Bud de deux manières. D'abord le moteur *Webdamlog* ajoute la possibilité d'évaluer des règles contenant des variables à la place des noms de relations et de pairs dans les règles. Puis afin de supporter la négation, *Webdamlog* permet aussi l'ajout et la suppression de règles dynamiquement, c'est à dire pendant l'exécution du programme. Enfin, je présente une technique d'optimisation basé sur la provenance pour la suppression des faits et des règles.

- Je présente l'architecture d'un pair *Webdamlog* contenant un moteur

d'évaluation *Webdamlog* et plusieurs adaptateurs[3] permettant au pair d'interagir avec les pairs non-*Webdamlog*. Je détails l'architecture et l'implémentation des lectures et mises à jour des faits et règles entre les adaptateurs et le moteur *Webdamlog*. La gestion des accès concurrents est basé sur le patron de conception[4] *Reactor pattern* [FMS09].

Je pense que ces contributions forment une bonne base pour résoudre les problèmes fréquemment rencontrés dans l'échange de données sur la toile, en particulier pour l'échange de données personnelles dans les réseaux sociaux.

Résumé de l'état de l'art

Cette thèse aborde deux domaines importants de l'informatique, les systèmes de données distribuées et l'inférence de connaissances.

Données distribuées Les systèmes distribués [ÖV99, AMR+11] sont des logiciels qui servent à coordonner les actions de plusieurs ordinateurs à travers l'envoi de messages. Ils sont caractérisés par les notions de consistance, de fiabilité, de disponibilité, de passage à l'échelle et d'efficacité. Dans le cas des bases de données, le système consiste en un ensemble de plusieurs bases de données, logiquement liées, distribuées sur un réseau d'ordinateurs. La distribution est transparente pour l'utilisateur : le résultat d'une requête ne dépend pas à priori du pair sur lequel elle a été posée. Sur la toile, la distribution est une composante de base de l'organisation du système. Le développement du langage commun XML et d'autres standards facilite l'expansion des échanges. Enfin, les systèmes pairs-à-pairs, structurés ou non, représentent l'aboutissement d'importants efforts de recherche en matière de distribution dans lesquels les nœuds ont des comportements extrêmement variés et flexibles.

L'inférence La connaissance est utilisée pour décrire la sémantique des données. La connaissance représentée dans des formats lisibles par l'humain comme sur Wikipedia par exemple est difficile à traiter par ordinateurs. Des systèmes d'intégration de la connaissance humaine en format interprétable par des machines est un sujet de recherche présenté dans [SKW07, LIJ+13]. Dans cette thèse, je m'intéresse surtout à la partie traitement des connaissances en format machine. Les fondements de l'inférence de connaissances reposent sur les bases de la logique mathématique, essentiellement des fragments de la

[3] wrappers
[4] design pattern

logique du premier ordre. Je m'intéresserai particulièrement aux systèmes déductifs de Hilbert. Ces systèmes sont basés sur des règles de Hilbert dans lesquels une règle déduit un nouveau fait à partir d'une conjonction de conditions sur un ensemble de faits déjà connus. Historiquement l'un des premiers langages à base de règles est Prolog [CR93] qui repose sur un algorithme d'évaluation nommé SLD [EK76]. Dans cette thèse je m'intéresse à datalog, un langage plus restreint que Prolog et qui offre de bonnes propriétés de terminaison. Datalog est un langage d'inférence particulièrement adapté aux bases de données qui supporte nativement la récursion contrairement à SQL le langage déclaratif habituellement utilisé dans les systèmes de gestion de base de données.

Le langage *Webdamlog* que nous présentons dans la section suivante est basé sur datalog avec négation [AHV95]. Principalement datalog muni de son extension distribuée [Hul89, NCW93, Hel10, GW10]. Dans les travaux précédent sur datalog, un programme positif est distribué sur plusieurs pairs après une phase de compilation. Nous nous intéressons à un déploiement beaucoup plus dynamique, et nous introduisons en particulier la notion de délégation.

Le langage *Webdamlog*

La gestion d'information distribuée est un problème important, en particulier sur la toile. Des langages basés sur datalog ont donc été proposé pour le modéliser. Nous introduisons ici un nouveau modèle, dans lequel des pairs autonomes échangent des messages et des règles (délégation). Nous étudions en particulier les conséquences sur l'expressivité de la délégation. Nous proposons aussi des restrictions du langage qui garantissent sa convergence.

Je présente un exemple où Alice souhaite gérer automatiquement l'organisation de ses réunion et plus particulièrement une conférence téléphonique qu'elle a noté dans son calendrier. Considérons un pair *Alice-phone*, avec la relation *calendar* qui contient l'agenda personnel d'Alice sur son téléphone et la relation *confMembers* correspondant à la liste des membres de la conférence téléphonique. Voici des exemples de faits:

at Alice-phone:
calendar@Alice-phone(confTel, 06/12/2013, Paris, Alice-phone)
confMembers@Alice-phone(Bob, agenda, Bob-laptop)

La règle suivante ajoute les entrées relatives à la conférence téléphonique du calendrier d'Alice dans ceux des autres membres de la conférence téléphonique:

at Alice-phone:
$calendar@$peer(confTel, $date, $place, Alice-phone) :-
calendar@Alice-phone(confTel, $date, $place, Alice-phone),
confMembers@Alice-phone($name, $calendar, $peer)

Il faut noter que les pairs et le nom des messages sont traités comme des données. La règle génère le nouveau fait suivant :

agenda@Bob-laptop(confTel, 05/12/2013, Paris, Alice-phone)

Le fait décrit un message envoyé d'Alice-phone à Bob-laptop. Ce fait extensionnel est consommé par Bob-laptop lorsqu'il le lit. Comme dans les bases de données déductives, le modèle distingue entre faits extensionnels et faits intentionnels. Par exemple, la relation *confMembers* peut être intentionnelle et définie ainsi :

at Alice-phone:
intentionnel confMembers@Alice-phone(string, relation, peer)
confMembers@Alice-phone($name, $relation, $peer) :-
contact@Alice-phone($name, $relation, $peer),
group@Alice-phone($name, confTel)

La sémantique du système est basée sur une sémantique locale, standard et sur l'échange de faits et de règles. Intuitivement, un pair donné calcule un nouvel état depuis son état courant en consommant ses faits locaux et en déduisant à partir de ses faits et de la sémantique locale les faits qu'il doit envoyer aux autres et à lui-même, ainsi que les règles qu'il doit déléguer aux autres. Un exemple de délégation est le suivant. Considérons la règle suivante:

at Bob-laptop:
confirm@$peer(confTel, $date, $place,Bob) :-
agenda@Bob-laptop(confTel, $date, $place, $peer),
checkAvailability@Bob-phone($date);

L'effet de la règle, étant donné le fait généré à l'intention de Bob-laptop, est d'installer la règle suivante sur le smartphone de Bob :

at Bob-phone:
confirm@Alice-phone(confTel, 05/12/2013,Paris,Bob) :-
checkAvailability@Bob-phone(05/12/2013);

Lorsque le smartphone de Bob exécute cette règle, en supposant que *confirm@Alice-phone* est extensionnel, si le fait

checkAvailability@Bob-phone(05/12/2013)

est satisfait le message suivant est envoyé à Alice:

confirm@Alice-phone(confTel, 05/12/2013, Paris,Bob)

Si *confirm@Alice-phone* est intensionnel, c'est la règle suivante qui est envoyée:

at Alice-phone:
confirm@Alice-phone(confTel, 05/12/2013, Paris,Bob) :-

Cette règle dont le corps est vide est toujours satisfaite sans condition et contrairement au fait précédent elle ne sera pas consommé par le pair Alice-phone cependant elle sera désinstallé à l'initiative de Bob-phone.

Sans rentrer dans les détails formels, il est intéressant d'étudier l'impact de la délégation sur l'expressivité du langage. En plus du langage général, noté WL, on peut distinguer deux sous-langages. Le premier, SWL, restreint la délégation aux vues. Le second, SWL, interdit complètement la délégation. Enfin, nous considérons les variantes autorisant les étiquetages temporels, notés WL^t, VWL^t et SWL^t respectivement. Les différences d'expressivité sont résumées sur la figure 1. Les inclusions sont strictes, à l'exception de celle de VWL^t dans VWL^t qui reste indéterminée.

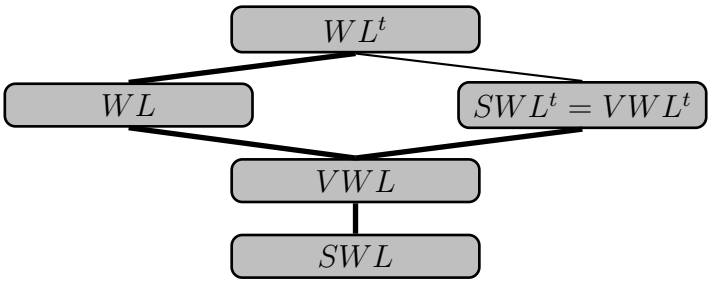


Figure 1: Expressivité des variantes de WL (les inclusions sont strictes quand l'arc est en gras)

Un autre point d'intérêt est la convergence du langage en fonction de l'ordre d'exécution des pairs. En règle générale, le résultat du calcul est à priori différent pour deux ordres d'exécution différents. Néanmoins, on peut isoler des cas monotones ou fortement stratifiés qui assurent la convergence, et ont une sémantique comparable à celle du cas où on centraliserait naturellement l'ensemble des faits et règles initiaux.

Le moteur de règles *Webdamlog*

Je considère la gestion de données distribuées sur la toile basée sur un réseau pair à pair d'acteurs autonomes et hétérogènes. Pour permettre aux pairs d'exprimer leurs propres tâches de gestion de connaissances tout en collaborant ensemble pour les tâches de gestion distribuées, je propose une implémentation d'un moteur d'évaluation du langage *Webdamlog* précédemment introduit.

Le moteur *Webdamlog* s'appuie sur un moteur d'évaluation de datalog distribué nommé *Bud* [ACHM11]. Le système *Bud* supporte efficacement les mises à jour et la distribution de datalog bien qu'il n'implémente pas la négation. *Bud* implémente l'algorithme d'évaluation semi-naïve pour l'inférence locale basée sur un système de déduction monotone positif de type chaînage avant. Afin d'évaluer un programme *Webdamlog* trois modifications majeurs au moteur ont été ajouté:

- Le support de règles contenant des variables à la place des noms de pairs ou de relations dans les règles.
- Le support des délégations, soit la réception de règles en plus de la réception de faits.
- L'ajout de règles pendant l'exécution du système.

De plus la sémantique particulière des relations extensionnelles de *Webdamlog* qui par défaut ne sont pas permanentes impose la redéfinition des structures de mise à jour de *Bud*.

J'introduis une série d'optimisations pour l'évaluation des programmes *Webdamlog*. Premièrement une optimisation basé sur l'échange différentiel pour les délégations. Deuxièmement je montre une technique d'optimisation du type Query-Subquery [Vie86] pour les règles distribuées. Enfin je propose un technique d'optimisation plus générale pour la gestion de la suppression dans un programme datalog avec mise à jour. Cette technique basé sur la provenance de la déduction en gardant le graphe de preuves des faits et règles déduits, je propage la suppression par une mise à jour du graphe. La technique d'évaluation standard recalcule l'ensemble des relations mises à jours en relançant l'algorithme d'évaluation semi-naïve. Dans le contexte extrêmement dynamique de *Webdamlog* ou les faits et les règles changent rapidement, cette optimisation est cruciale afin d'obtenir des performances raisonnables de la part du système. Je présente dans la section 4.5 une série d'expérimentation permettant de valider mes optimisations à large échelle.

L'architecture du pair *Webdamlog*

La gestion d'information sur Internet s'appuie sur une grande variété de systèmes spécialisés pour des tâches particulières. Dans l'exemple préalablement introduit, *Joe* souhaite interagir avec de nombreux services Web distants. Certains systèmes proposent des adaptateurs[5] pour intégrer les données en un unique point centralisé et ainsi permettre à *Joe* de gérer automatiquement ses données via une unique interface. Je présente un pair *Webdamlog* qui muni des adaptateurs nécessaires, permet de collaborer avec les différents services tout en gérants les données *en place*, c'est à dire en conservant la distribution des données auquel *Joe* est habitué. La nécessité de ne pas se reposer sur un unique prestataire auquel il serait nécessaire de confier toutes ses données personnelles me semble être la motivation majeur pour l'utilisation d'un système tel que *Webdamlog*.

Dans ce chapitre je décris l'architecture et l'interaction d'un pair *Webdamlog* avec les autres pairs non-*Webdamlog*. L'intégration d'adaptateurs autour du moteur de déduction *Webdamlog* permettent de fourni des fonctionnalités nécessaires tels qu'un interface graphique pour les interactions avec l'utilisateur, une base de données pour le stockage persistent des données et d'autre adaptateurs pour la communication par courriels[6] ou avec un réseau social comme Facebook.

Je définis un modèle général pour la gestion des évènements autre que les faits et règles *Webdamlog* basé sur le patron de conception[7] *Reactor pattern* [FMS09]. Puis je présente l'interface de programmation[8] pour les adaptateurs d'un pair *Webdamlog*. Un exemple d'application réalisable grâce au système *Webdamlog* a été présenté lors d'une démonstration [4] à SIGMOD 2013. J'ai implémenté un système de réseau social pour le partage de photos lors d'une conférence. L'interface de base de cette application permet de lister ses photos, ajouter des annotations, des notes et des commentaires. Les participants étaient invités à lancer leur propre pair avec leurs propres photos. Je leur montrais comment grâce à un petit nombre de règles *Webdamlog*, l'application permet de découvrir les autres membres de la conférence et distribuer ses photos avec ses amis. Lors de la démonstration je proposais aux participants de modifier leur pair pour y ajouter des règles permettant de modifier le comportement de l'application afin de réaliser automatiquement des tâches personnalisées.

[5] wrappers
[6] emails
[7] design pattern
[8] Application Programming Interface (API)

Étude utilisateur

Dans cette thèse, je mets en avant l'utilisation du langage *Webdamlog*, un langage déclaratif qui permet d'abstraire les détails techniques de la distribution de la connaissances pour permettre à l'utilisateur de se concentrer sur la spécifications des tâches. Lors de cette étude, nous avons réuni un échantillon d'utilisateurs informaticiens et non informaticiens et de divers niveaux d'études pour tester leurs capacités à utiliser le langages *Webdamlog*. Nous leur avons présenté un cours de 20 min visant à enseigner les bases du langage, puis nous leur avons fait passer un test. Les exercices du test visaient à évaluer le niveau de compréhension de petits programmes *Webdamlog* puis leurs capacités à écrire eux même des règles permettant d'accomplir automatiquement des tâches typiques qu'une application utilisant *Webdamlog* permet d'accomplir.

Conclusion

La philosophie de *Webdamlog* est de permettre de redonner le contrôle de ses données aux utilisateurs de la toile. Alors que le courant actuel nous pousse à confier de plus en plus nos données à des sociétés tierces essentiellement via l'infonuagique[9], *Webdamlog* insiste sur la devise "Faites le vous même"[10], c'est à dire gérez vos propres données avec vos propres systèmes. Grâce au concept de délégation, le langage *Webdamlog* permet l'automatisation de tâches complexe de gestion de données distribuées, et en particulier celles qui requièrent la collaboration de plusieurs systèmes hétérogènes. Contrairement aux systèmes centralisées propriétaires, le code de la toile est ouvert[11], à l'instar de *Webdamlog* qui est basé sur le partage du code.

Webdamlog ouvre un grand nombre de directions de recherche. Pour conclure cette thèse, je mentionne quelques directions qui selon moi sont les plus importantes:

- Une étude utilisateur approfondie de l'utilisation de *Webdamlog* par les utilisateurs courants de la toile, c'est à dire ceux n'ayant que peu de connaissances de l'informatique. Il semble essentiel de comprendre les possibilités et les limitations de notre approche.

- Il serait intéressant de développer de meilleurs interfaces pour simplifier

[9] cloud computing
[10] Do it yourself
[11] open-source

la conception d'application pour faciliter la prise en main par les futurs développeurs.

- Le contrôle d'accès pour les programmes *Webdamlog* est une pierre angulaire manquante dans notre système. Cette voie de recherche est la plus importante des priorités qui permettrai le développement de réel applications.

- *Webdamlog* encourage le partage de connaissances entre les pairs ou à l'intérieur d'une communauté. Certainement que de tels échanges seraient facilité par l'amélioration de *Webdamlog* avec les technologies d'ontologies du Web sémantique.

- Finalement, nous avons montré comment améliorer les performances en utilisant certaines techniques d'optimisation. Il faudrait investir plus amplement dans ce domaine pour passer à l'échelle de la toile[12], essentiellement en terme de nombre de pair, de charge de traitement et de taille des données.

[12] Web

Chapter 1

Introduction

Information management on the Internet relies on a wide variety of systems, each specialized for a particular task. The personal *data* and favorite *applications* of a Web user are typically distributed across many heterogeneous devices and systems, e.g., residing on a smartphone, laptop, tablet, TV box, or managed by Facebook, Google, etc. Additional data and computational resources are also available to the user from relatives, friends, colleagues, possibly via social network systems. Because of the distribution and heterogeneity, the management of personal data and knowledge has become a major challenge.

A Web user is regularly facing information management tasks that may be extremely cumbersome to carry out manually. Yet, automating these tasks, for example by writing scripts, is far beyond the skills of most Web users. Some systems attempt to provide integrated services to support these needs. For instance, Facebook provides a wrapper service to integrate Dropbox accounts and blogs. However, such services are often limited in the functionality they support. Also, by delegating such services to systems like Facebook, a user is lead to entrust more and more of his data to a single company, at the cost of losing ownership and control of his own data.

Our goal is to enable a Web user to easily specify distributed data management tasks *in place*, i.e. without centralizing the data to a single provider. Our system is therefore not a replacement for Facebook, or any centralized system, but an alternative that allows users to launch their *own peers* on their machines with their own personal data, and to collaborate with Web services.

Towards this goal, we propose *Webdamlog*, an elegant language for managing distributed data and knowledge. As a datalog-style language, its main benefits are the familiar ones: a declarative approach alleviates the conceptual complexity on the user, while at the same time allowing for powerful performance optimizations on the part of the system. Besides this language,

our contributions consist of the design and implementation of an engine supporting *Webdamlog*, novel optimization techniques taylored to this setting, and the development of an environment for the peers supporting *Webdamlog*.

Language *Webdamlog* is a datalog-style language that emphasizes cooperation between autonomous peers communicating in an asynchronous manner. The language extends datalog in a number of ways, supporting updates, negation, distribution and importantly *delegation*, a novel feature allowing peers to exchange not only facts but also rules. We present a limited user study that demonstrates the usability of the language, i.e., that users can use the language after a minimal amount of training.

Engine We designed and implemented a *Webdamlog* engine. The engine extends a distributed datalog engine, namely *Bud*, with the support of delegation and of a number of other novelties of *Webdamlog* such as the possibility to have variables denoting peers or relations. To support very dynamic environments where the knowledge of peers vary rapidly notably by acquiring new rules from other peers via delegation, we introduce novel techniques, notably one based on the provenance of facts and rules. We present experiments that demonstrate that the rich features of *Webdamlog* can be supported at reasonable cost and that the engine scales to large volumes of data.

Peer A *Webdamlog* peer provides an environment for the engine. In particular, it supports wrappers to exchange *Webdamlog* knowledge with non-*Webdamlog* peers. We illustrate these peers by presenting a picture management application that we used for demonstration purposes. In this application, users can communicate through a Web interface, between them by mail, with Facebook, and store data in a database.

Organization The thesis is organized as follows. We first discuss the state or the art in Chapter 2. In Chapter 3, we introduce the *Webdamlog* language. We present the engine, the main optimization techniques, as well as experiments, in Chapter 4. The peers and the picture management application are covered in Chapter 5, and the user study in Chapter 6. We conclude with Chapter 7.

Chapter 2

State of the Art

The two main aspects of this thesis are *distributed information* and *inference*. We next give an overview of these two topics in the area of data management. To conclude the section, we mention *Webdam exchange*, a system for distributed data management that influenced the work presented here.

2.1 Distributed Information Systems

Distributed information systems are now a well developed area of computer science, covered by a large number of reviews and books. e.g., [AMR+11, ÖV99]. In the following discussion, we consider its most relevant aspects for this thesis. We first present the general aspects of distributed systems, then review successively databases, Web data and peer-to-peer distribution.

2.1.1 Distributed systems

[AMR+11] defines a distributed system as some software that serves to coordinate the actions of several computers. This coordination is achieved by exchanging messages, i.e., pieces of data conveying information. The system relies on a network that connects the computers and handles the routing of messages.

Distributed systems are characterized by the following desirable properties:

- *Consistency* [DHJ+07] denotes the ability of a distributed system to give the same answer to a client regardless of the server it is connected to.

- *Reliability* [Bir05] denotes the ability of a distributed system to experience no failure in any given time interval.

- *Availability* [ÖV99] denotes the ability of a distributed system to be operational at a given point in time.
- *Partition tolerance* denotes the ability of a distributed system to operate despite arbitrary message loss or failure of part of the system.
- *Scalability* [MMSW07] denotes the ability of a distributed system to continuously evolve in order to support a growing amount of tasks and data. In general, one is interested by linear scalability, i.e., a growing of the system resources proportional to that of the tasks and data.
- *Efficiency* denotes the ability of a distributed system to minimize the response time (when the first item is delivered) and to maximize the throughput (the number of items delivered by unit of time).

One is typically facing a trade-off between these properties. In particular, the CAP theorem [GL02] states that a distributed system cannot provide simultaneously consistency, availability and partition tolerance. This last result is of particular importance for us, since we aim at providing some precise results for our system, but also consistency guarantees.

2.1.2 Distributed databases

[ÖV99] defines a distributed database as a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DBMS) is then defined as the software system that permits the management of the distributed database and makes the distribution transparent to the users [LM75, SW85]. It provides a shared structure among the data, and an access via a common interface. Distributed DBMSs are intended to provide data independence, network transparency, replication transparency and fragmentation transparency. Usually DBMSs improve reliability and availability by replicating components, thereby eliminating single points of failure or bottleneck, while letting the user ignore distribution issues.

[ÖV99] describes the architecture of a distributed DBMS by characterizing the autonomy of local systems (tight integration, semi-autonomy and total isolation), their distribution (no distribution, client-server or peer-to-peer) and their heterogeneity (homogeneous or heterogeneous system).

2.1.3 Data on the Web

With the development of *Internet* [RFC74] and *HTML* [W3C13], the *Web* [BLC90] rapidly became an essential way of data distribution. This position

was further strengthened by the development of *XML* [W3C08a, AQM+97] relaxing the rigid relational data structure, that highly eases exchange and integration of heterogeneous data in a semi-structured way. The World Wide Web Consortium, that is in charge of promoting and developing XML usage, proposed a wide range of standards and the research community has been particularly active on different topics including typing [W3C04b], querying [W3C10, BKS02] or transforming XML [W3C99, AKSS09, ABM09]. There is now a large number of books surveying aspects of Web's data. See, e.g., [AMR+11].

As his founder Tim Berners-Lee foresaw, the Web is also developing a layer of semantics on top of XML or HTML, using *ontology* languages such as RDF [W3C04a] and OWL [W3C09] to facilitate data integration. More formal analysis of these languages can be found in [AH08, AvH08]. Integration also benefits of the large amount of work on *mediation* [HZ96]. See [AMR+11] for a survey. This leads to the domain of *knowledge bases* centered around the management of knowledge in machine-processable formats. That is the topic of Section 2.2.

Finally, the development of *Web services* gave an infrastructure for distributed Web data management. This infrastructure is based on XML standards such as SOAP [W3C07a], WSDL [W3C07b] and UDDI [OAS04] which respectively normalize the structure of data to exchange as objects ; describe the methods provided for the previous objects ; and specify communication with Web services. Other additional standards are used to express complex operations using multiple Web services such as service *workflows* [HNN09, NC03] with BPEL [OAS07] and *orchestration* of services with WSCL [W3C02] to thereby achieve collaboration of autonomous entities on the Web. Some models such as *ActiveXML* [ABM08, ABCM04, ABMG10] aim at providing a formal model for intensional data that is the data obtained by service calls on the Web and distributed data intensive applications.

To summarize, the Web is now a standard way of sharing and managing data. Our work, as part of the *Webdam* Project [ERC13], focuses on providing better foundations for collaboration of autonomous peers. The *Webdam system* relies on standard models and tools.

2.1.4 Peer-to-peer systems

A *peer to peer* (P2P) network (See, e.g., the surveys in [TS04, Wal03]) is a large network of nodes, called *peers*, that are both clients and servers and that are willing to cooperate in order to achieve a particular task. It is a particular kind of distributed systems that assumes that the organization of the nodes is loose and flexible. Indeed, the peers are highly autonomous, choosing when

they participate to the network and how much resource (CPU, memory, ...) they provide to the system. It is also often assumed they use an *overlay network*, i.e., a graph of connections laid over a physical infrastructure, e.g., the Internet.

A general search technique on this kind of networks is *flooding*: a peer disseminates its request to all its friends, that flood in turn their own friends. One may also use other forms of *gossiping*, for example by choosing randomly only a small number of friends to propagate the request. Such P2P networks are called *unstructured*. There are more structured ways for searching for information in the network (*structured P2P networks*), based on access structures such as distributed hash tables [Lit80, LNS96, KLL+97, DHJ+07] or distributed search trees [LNS94, KW94, JOV05, CDG+08].

Since focus in this thesis is on a P2P system, the *Webdam system* can be built on all these different distribution policies in a unified manner. The goal is to facilitate the collaboration of autonomous peers towards solving content management tasks. A number of works have argued for developing a holistic approach to distributed content management, e.g. *P2P Content Warehouse* [Abi03], *Dataspaces* [FHM05] and *Data rings* [AP07a]. Such situations arise for instance in personal information management, that is often given as an important motivating example [FHM05].

2.1.5 Social networks

Webdamlog was initially motivated by the idea of the management of personal data in social networks. Contrary to the most famous social networks that are entirely centralized, we wanted to manage data in a peer to peer way as motivated in [AP07b]. Switching from one authoritative server to a collaborative set of untrusted peers [NCR08, KBC+00] raises issues about privacy. Trust in peer-to-peer environments where one frequently encounters unknown agents is addressed in [AD01, YHY07], anonymization in [CSWH01], encryption and access control in [MS02, MS03, WL82] and [KGG+06] proposed a distributed identity management with access control based on the social network of users. In particular, it uses the standard Friend-Of-A-Friend (FOAF) [BM10] representation of the social network. An access control policy model based on the social network and trust has also been proposed by [AVM07, AGP11]. Finally, [BSVD09, Dia] proposes an implementation of a peer-to-peer social network based on a distributed hash table and addresses privacy issues. Such approaches require that a particular program, predetermined for the application, is deployed identically on each peers. To the best of our knowledge, there are few works about heterogeneous [Kol05] and customizable peer [MZZ+08, RS09] in a collaborative environment that would

behave differently according to the user needs. Recently works on dynamic and adaptive programs based on rules have been achieved in particular for ontology languages [Kif08, BAP+12].

2.1.6 Contribution

The focus of my thesis is on peer-to-peer architecture rather than client-server communications usually used in centralized systems. We consider very heterogeneous data and a total autonomy between the components of the system. This is the main contribution of our *Webdamlog* system, the *collaboration* of such autonomous peers managing their own data *in place*. Hence the current *Webdamlog* system has been strongly influenced by ActiveXML although the XML trees have been abandoned for traditional relational data structures to simplify and to be able to focus on other issues, notably inference.

In the next section we consider the topic of knowledge bases and inference since the *Webdam system* is a distributed knowledge base system.

2.2 Knowledge bases

Knowledge can be used to describe the semantic of data. Two kinds of knowledge formats can be considered:

- human-readable knowledge e.g. Wikipedia that is usually read and updated manually by humans.
- knowledge in machine-readable format e.g. Yago [SKW07] on which searches and updates can be automatically performed by machines.

Systems transforming one type of knowledge base in another, are presented in [SKW07, LIJ+13] and integration of different knowledge bases in [AMR+11]. These problems will not be considered in this thesis and the focus is on machine-readable knowledge.

2.2.1 Processing knowledge

Knowledge in machine-readable format typically relies on some mathematical logic as its foundation and is processed by an inference system guided by logical reasoning. The logic is usually a fragment of first order logic it that serves as basis for query languages. The formalism of deductive systems can be natural deduction, sequent calculus, tableaux method, resolution or

Hilbert-style deductive systems which will be our focus in the following of the thesis.

The programming languages implementing these formalisms are rule-based languages. The concept of rules relies on the basic notion of conditional branching or the "if ... then ..." construct. The then part is processed only if the if part holds. Rule systems use a notion of *predicates* that holds or not to represent the raw data. E.g. the fact that two people are friend may be represented with a *predicate* friend as friend(Alice,Bob). Using variables, represented by a dollar prefixed letter, a rule could be:

Rule: if friend($x,$y) then like($x,$y)

It represents some knowledge added to our data. A rule-based system that understands this rule derives that all pairs of friend like each others. A rule-based system is a particular implementation of the syntax and semantics of rules which may be extended in a number of ways e.g. with existential quantification, disjunction, negation and functions.

Historically, Prolog [CR93] is considered as one of the first and the most expressive rule-based language ; however a main flaw is to not be declarative, e.g. because of the cut operator and because the order of clauses matters in the evaluation. It is based on SLD resolution, a top-down technique for deductive system [EK76].

In Section 2.1.3, we mentioned that Web data are often described with ontologies that are fragments of first order logic, on which deduction systems apply [CGL09]. In the context of the Web, considering reasoning in a distributed manner is crucial as discussed in [ACG+06]. See [AMR+11, FHMV03] for more details on ontology languages reasoning.

We discuss next the family of datalog languages.

2.2.2 Datalog

In datalog the previous rule is written:

like($x,$y) :- friend($x,$y)

with the left-hand side part of the operator ":-" called the head and the right-hand side called the body. Following the "if ... then ..." structure, rules are read: if body holds then head is derived. The datalog semantic imposes that all variables in the head appear in the body. Basic datalog extends this structure with:

conjunction of atoms in the body: like($x,$y) :- friend($x,$y), friend($y,$x)
both facts should be true to derive the head.

disjunction that is the same fact can be derived from different conditions. Datalog program allows multiple rules with the same head. For example, the program:

```
like($x,$y) :- friend($x,$y)
like($x,$y) :- friend($y,$x)
```

means x likes y if x is a friend of y or y is a friend of x.

recursion by allowing the same predicate in the head and the body of the same rule: `friend($x,$y) :- friend($x,$z), friend($z,$y)` is the classic transitive closure which means that everybody is friends with the friends of its friends.

A datalog programs P is a set of rules. A set of facts are gathered in an extensional database noted I as instance. In brief, the semantics of a datalog program is the minimal fixpoint reached when we cannot deduce any new facts by applying P on I. These derived facts are called intensional. The union of the extensional and intensional facts represent the whole facts considered to be true ; everything else is false. This is the close-world assumption contrary to some other rule-based languages such as OWL that makes an open-world assumption.

Datalog is also often extended with negation, denoted datalog$^{\neg}$. Negation and recursion together raise a number of issues. For instance,

- For $I = \{p\}$ and $P = \{p \text{ :- } \neg p\}$, there is no fixpoint
- For $I = \varnothing$ and $P = \{p \text{ :- } \neg q; \quad q \text{ :- } \neg p\}$, there are two minimal fixpoints $\{p\}$ or $\{q\}$

This leads to defining different semantics for the negation e.g. stratified or well-founded semantics.

Datalog also has a non-monotonic extension noted datalog$^{\neg\neg}$ to specify that negation can occur in the body and in the head of rules. This is a convenient way to handle deletion of facts. The language datalog$^{\neg\neg}$ is in the spirit of active databases, and since it allows to use extensional predicates in the head of the rules.

Datalog has been the subject of a large amount of works in the database community ; see [AHV95]. Basically, datalog enhances the classical relational calculus and algebra, that are at the foundation of SQL, with recursion. Although recursion has been added in SQL3 [ISO99], datalog natively supports recursion with an elegant syntax. The full description of the semantic, and

evaluation of datalog following the bottom-up semi-naive algorithm is given in [AHV95] along with the description of adding negation to datalog. And discussions on datalog and first order logic expressivity are given in [AG94].

Alternatives to datalog-like languages for data management based on rules have been proposed. For instance:

- F-logic [KLW95], an object oriented language for data and knowledge representation.
- HiLog [CKW93], a higher-order programming language that uses functions as values as in lambda calculus.

Both are implemented in the Flora system [YKZ03] using an alternative inference system based on tabling-logic [YK00].

The next section focuses on distributed versions of datalog engines.

2.2.3 Distributed datalog

The *Webdamlog* language participates in the renewed interest in datalog, see [Dat10]. In particular distributed datalog allows to use remote atoms in the head of rules to communicate via the network. The elegant syntax of datalog for recursion is essential when graph data are considered. This is the case for instances in declarative networks as shown in [AKBC+12, LCG+06, ZFS+11, ZST+10], in the implementation of the two-phase-commit protocol in [Int12], or in sensor networks communications [GW10, AKGU12] that present an original top-down evaluation algorithm for distributed datalog.

To our knowledge, the first attempts to distribute datalog on different peers are [Hul89] and [NCW93]. The first distributes a positive datalog program on different machines after a compilation phase. The second adapts classical transformations of positive programs based on semi-joins to minimize distribution cost. Perhaps the work closest to the *Webdamlog* language is [AAHM05b] that adapts query-subquery optimization [Vie86] to a variant of positive distributed datalog. We will also be interested in negation, in particular by stratified negation [CH85], and by active rules in the style of datalog$^{\neg\neg}$ [AV91, AGM08, BCGR98].

The most interesting use of datalog-style rules for distributed data management came recently from the Berkeley and U. Penn database groups. They used distributed versions of datalog to implement Web routers [LHSR05], DHT [LCH+05] and Map-Reduce [ACC+10] rather efficiently. By demonstrating what could be efficiently achieved with this approach, these works were essential motivations for our own. The most elaborate variant of distributed datalog used in these works is presented in [LHSR05, LTZ+09, MHB+10, CCHM08]

and formally specified in [NR09, PRS09, MAC+12]. In these papers, the semantics is operational and based on a distribution of the program before the execution. In view of issues with this model, a new model was recently introduced in [Hel10], based on an explicit time constructor. The semantics of negation together with the use of time in that model seems rather unnatural. In particular, time is used as an abstract logical notion to control execution steps and the future may have influence on the past. As a consequence, we found it difficult to understand what applications are doing as well as to prove results on their language. The development of *Webdamlog* reuses most of the *Bud* [ACHM11] inference engine from Berkeley that has been proven to be efficient.

2.2.4 Provenance and optimization

The need of a logic language for knowledge representation and especially for access control on data is formalized in [Aba09] and implemented in declarative systems as [BFG07, Bry05]. However inference systems bring their own intrinsic security issues. As described in [FJ02], it is difficult to control indirect data disclosure via inference. Access control in distributed environment was a prime motivation in a previous model called *Webdam exchange* discussed in 2.3. Access control will not be considered in this thesis. It is left for future work.

In this thesis, we record provenance of knowledge to optimize deletion and maintain efficiently *Webdamlog* program evaluation. However it has been considered for different purpose: for access control [GKT07, KIT10], for security policies [MFF+08] or to synchronize distributed data [GKIT10, GKIT07]. See [BT07], for a general presentation and challenges around data provenance. Maintaining provenance of knowledge from inference system is considered in [ZFS+11, ZST+10]. Works around fine grained provenance on workflows [ADD+11] as an optimization for deletion inspired us for our system.

2.2.5 Contribution

Our main concern in designing *Webdamlog* has been to provide an elegant and unified way to allow each peer to manage personal data according to the preference of the user. We considered typical Web users that may have distributed their data on several locations and services. The declarative nature of our language *Webdamlog*, based on a distributed datalog allows to alleviate the complexity of managing the distribution of the data. The most striking novelty of *Webdamlog* is to allow the distribution of the knowledge

via a new feature we developed for *Webdamlog*, namely the *delegation* of rules. The declarative approach of *Webdamlog* also allows us to provide optimization mechanisms for *Webdamlog* evaluation.

2.3 Webdam exchange

The work developed in this thesis is a continuation of the thesis of Alban Galland [Gal11] that lead to designing the *Webdam exchange* model [AGP11] and to the development of the *Webdam exchange* system [6] that we briefly discuss next.

In a demonstration of a system called *Webdam exchange* [6], we addressed the problem of access controls in peer to peer environment. The peers were running standard Java application. The basics of the *Webdam exchange* system were to be able to authenticate the peer who requests some data and confront it to an access control list (ACL) to grant or refuse access. For each relation, a list of reader, writer and owner where defined by the owner and only these principals could perform these actions.

Model In social networks, users bring data to the network and are willing to share with others, but also wish to control what portions of the data can be viewed or updated by others. Users would also like to access and update information if desired and entitled to. This is the setting of the *Webdam exchange* model that aims to achieve access control of personal data in peer to peer environment with the same level of security as in centralized systems. It also leverages and accommodates a wide variety of authentication systems already available on the Web.

For access control, three kinds of meta-data, namely access control list, secret, and hint are kept for each fact. Using these meta-data, *Webdam exchange* shows how to describe access control mechanisms based on authenticated provenance for different security protocols such as asymmetric cryptographic keys or HTTP access controlled by login/password. It also describes how to exchange information between peers that are trusted or untrusted, in clear or encrypted communications.

System The data model of the WebdamExchange system is a direct translation in XML of the WebdamExchange model. It uses Java XML Binding (JAXB) technology to construct a direct equivalence between Java classes and their XML representation, used for Web service communications, encryption and serializations that fit the Web standards. The main contribution of the *Webdam exchange* system, was the design of a modular architecture to keep

communication, encryption, security policy and storage system independent of each other. Hence it allows to describe in the security policy, according to meta-data statements, which kinds of communication, encryption or storage to use.

At the dawn of Webdamlog *Webdam exchange* and *Webdamlog* are both addressing the problem of personal data management in peer to peer environments with a strong emphasis on access controls in *Webdam exchange*. They both deal with the heterogeneity of personal user preferences to manage its data.

Nevertheless there is a fundamental difference that comes from the fact that *Webdam exchange* applications are hard coded in plain Java code contrary to *Webdamlog* systems that relies on the declarative language *Webdamlog* to describe their behavior. The main motivation for that is that typical users don't want to write complicated programs. In the following chapters, we will describe how *Webdamlog* brings a powerful mechanism called delegation that enhanced collaboration.

Also *Webdam exchange* data model strongly relies on trees and especially nested structure to keep chains of provenance and authentication needed to enforce provenance, while *Webdamlog* is based on relations. Both could be combined as it would not be difficult to introduce trees in *Webdamlog* language. However from a system viewpoint this would mean a very different implementation.

Chapter 3

Webdamlog language

The management of modern distributed information, notably on the Web, is a challenging problem. Because of its complexity, there has recently been a trend towards using high-level Datalog-style rules to specify such applications. We introduce here a model for distributed computation where peers exchange messages (i.e., logical facts) as well as rules. We consider peers as any kind of system with computing capabilities and network connection to capture the heterogeneity of the agents on the web e.g. a laptop, a smartphone, or a computer cluster in a DHT. The model provides a new setting with a strong emphasis on dynamicity and interactions (in a Web 2.0 style). Because the model is powerful, it brings a clean basis for the specification of complex distributed applications. Because it is simple, it gives a formal framework for studying many facets of the problem such as distribution, concurrency, and expressivity in the context of distributed autonomous peers.

As mentioned in the previous chapter, there has been renewed interest in studying languages in the Datalog family for a broad range of applications from program analysis, to security and privacy protocols, natural language processing, or multi-player games. For references, see [Hel10] and the proceedings of the Datalog 2.0 workshop [Dat10]. Here, we are concerned with using rule-based languages for the management of data in distributed settings, as in Web applications [ABM04, ASV09, FMS09, ABGR10], networking [LCG+06, LMO+08, GW10] or distributed systems [LCG+09]. The arguments in favor of using Datalog-style specifications for complex distributed applications are the familiar ones. See, e.g., [Hel10].

We propose a new model for distributed data management that combines, in a formal setting, deductive rules as in Datalog with negation [CH85] (to specify intensional data) and active rules as in Datalog$^{\neg\neg}$ [AV91] (for updates and communications). There have already been a number of proposals for combining active and deductive features in a rule-based language; see [LLM98,

Lud98, Hel10] and our discussion of related work. However, there is yet to be a consensus on the most appropriate such language. We therefore believe that there is a need to continue investigating new language features adapted to modern data management and to formally study the properties of the resulting new models.

The language we introduce, called *Webdamlog* is presented in [3], it is tailored to facilitate the specification of data exchange between autonomous peers, which is essential to the applications we have in mind. Towards that goal, a new feature we introduce is *delegation*, that is, the possibility of installing a rule at another peer. In its simplest form, delegation is essentially a *remote view*. In its general form, it allows peers to exchange rules, i.e., knowledge beyond simple facts, and thereby provides the means for a peer to delegate work to other peers, in Active XML style [ABM08]. We show using examples that because of delegation, the model is particularly well suited for distributed applications, providing support for reactions to changes in evolving environments.

A key contribution of this chapter is a study of the impact of delegation on expressivity. We show that view delegation (delegation in its simplest form, allowing only the specification of views) strictly augments the power of the language. We also prove that full delegation further increases it. These results demonstrate the power of exchanging rules in addition to facts.

A message sent from peer p, received at peer q, that starts some task at q, introduces a kind of synchronization between the two peers. Thus, time implicitly plays an important role in the model. We show that when explicit time is allowed (each peer having its local time), view delegation no longer increases the expressive power of the language.

Because of their asynchronous nature, distributed applications in *Webdamlog* are nondeterministic in general. To validate our semantics for deductive rules, we study two kinds of systems that guarantee a form of convergence (even in presence of certain updates). These are positive systems (positive rules and persistence of extensional facts) and strongly-stratified systems (allowing a particular kind of stratified negation [CH85] for restricted deductive rules and fixed extensional facts). We also show that both types of systems essentially behave like the corresponding centralized systems.

Organization The chapter is organized as follows. We introduce the model in Section 3.1, first by means of examples and then formally. In the following section, we discuss some key features of the model and illustrate them with more examples. In Section 3.3, we compare the expressivity of different variants of the language. In Section 3.4, we discuss the convergence of *Webdam-*

log systems and compare the semantics to the "centralized semantics", for the positive and strongly-stratified restrictions of the language. In Section 4.3, we mention optimization techniques. The final section concludes with directions for future work.

3.1 Model of data

In this section, we first illustrate the model with examples, then formalize it. More examples and a discussion of key issues will be provided in the next section.

3.1.1 Informal presentation

We introduce with a first example the main concepts of the model: the notions of *fact* that captures both *local tuples* and *messages* between peers, of *extensional* and *intensional* data, and of *Webdamlog* rule.

Consider a particular peer, namely *Alice-phone*, with the relation *calendar* that gives the calendar entry that Alice entered from her phone and the relation confMembers that gives the list of members of the conference call and how to send them calendar invitation (on which servers, with which messages). Examples of facts are:

at Alice-phone:
calendar@Alice-phone(conference, 06/12/2013, Paris, Alice-phone)
confMembers@Alice-phone(Bob, agenda, Bob-laptop)

The following rule, called [*Send-Invitation*] , is used to include conference call entries from Alice's agenda into the agendas of other members of the conference call, and in particular into Bob's agenda:

at Alice-phone:
$calendar@$peer(conference, $date, $place, Alice-phone) :-
calendar@Alice-phone(conference, $date, $place, Alice-phone),
confMembers@Alice-phone($name, $calendar, $peer)

Observe that peer and message names are treated as data. The two previous facts represent pieces of local knowledge of *Alice-phone*. Now consider the new fact generated by the rule:

agenda@Bob-laptop(conference, 06/12/2013, Paris, Alice-phone)

This fact describes a message that is sent from *Alice-phone* to *Bob-laptop*.

As in deductive databases, the model distinguishes between extensional relations that are defined by a finite set of ground facts and intensional relations that are defined by rules. So for instance, the relation *confMembers* on Alice-phone may be intensional and defined as follows:

```
at Alice-phone:
intensional confmembers@Alice-phone(string, relation, peer)
  confmembers@Alice-phone($name, $relation, $peer) :-
      contact@Alice-phone($name, $relation, $peer),
      group@Alice-phone($name, conf)
```

Observe that it is defined using extensional relations.

As usual, intensional knowledge is defined by rules such as the previous one, that we call *deductive* rules. Other rules such as the [*Send-Invitation*] rule, that we call *active*, produce extensional facts. Such an extensional fact is received by the peer (e.g., Bob-laptop and Alice's phone). During its next phase of local processing, this peer will consume these facts and produce new ones. By default, any fact that has been processed disappears. Facts can be made persistent using persistence rules, illustrated next on the relation *calendar@Alice-phone*:

```
at Alice-phone:
  calendar@Alice-phone($name, $date, $place, $peer) :-
     calendar@Alice-phone($name, $date, $place, $peer),
     ¬ del.calendar@Alice-phone($name, $date, $place, $peer)
```

The rules state that in this relation *calendar* a fact persists unless there is explicitly a deletion message (e.g., *del.calendar*).

Delegation by example

In the model, the semantics of the global system is defined based on local semantics and the exchange of messages and rules. Intuitively, a given peer chooses how to move to another state based on its local state (a set of personal facts and messages received from other peers) and its program. A move consists in (1) consuming the local facts, (2) deriving new local facts, which define the next state, (3) deriving nonlocal facts, i.e., messages sent to other peers, and (4) modifying their programs via "delegations".

The derivation of local facts and messages sent to other peers are both standard and were illustrated in the previous example. The notion of delegation is novel and is illustrated next. Consider the following rule, installed at peer *Bob-laptop*:

at Bob-laptop:
confirm@$peer(conference, $date, $place, Bob) :-
calendar@Bob-laptop(conference, $date, $place, $peer),
checkAvailability@Bob-phone($date);

where *calendar@Bob-laptop*, *checkAvailability@Bob-phone* and *confirm@Alice-phone* are all extensional. Its semantics is as follows. Suppose that *calendar@Bob-laptop(conference, 06/12/2013, Paris, Alice-phone)* holds, then the effect of this rule is to install at *Bob-phone* the following rule:

at Bob-phone:
confirm@Alice-phone(conference, 06/12/2013, Paris, Bob) :-
checkAvailability@Bob-phone(06/12/2013);

The action of installing a rule at some other peer is called *delegation*. When *Bob-phone* runs, if *checkAvailability@Bob-phone(06/12/2013)* holds, it will send the message *confirm@Alice-phone(conference, 06/12/2013, Paris, Bob)* to *Alice-phone*.

Now suppose instead that *confirm@Alice-phone* is intensional. When *Bob-phone* runs, if *checkAvailability@Bob-phone(06/12/2013)* holds, the effect of this rule is to install at *Alice-phone* the following rule:

at Alice-phone:
confirm@Alice-phone(conference, 06/12/2013, Paris, Bob) :-

The intuition for the delegation from *Bob-laptop* to *Bob-phone* is that there is some knowledge from *Bob-phone* that is needed in order to realize the task specified by this particular rule. So, to perform that task, *Bob-laptop* delegates the remainder of the rule to *Bob-phone*. The delegation from *Bob-phone* to *Alice-phone* is somewhat different. Peer *Bob-phone* knows that *confirm@Alice-phone* (an intensional fact) holds until some change occurs. As *Alice-phone* may need this fact for his own computation, *Bob-phone* will pass this information to *Alice-phone* in the form of a rule (since as a fact, it would be consumed).

We next formalize the model illustrated by the previous example.

3.1.2 Formal definitions

Alphabets

We assume the existence of two infinite disjoint alphabets of sorted *constants*: *peer* and *relation*. We also consider the alphabet of *data* that includes in

addition to *peer* and *relation*, infinitely many other constants of different sorts (notably, *integer*, *string*, *bitstream*, etc.). It is because *data* includes *peer* and *relation* that we may write facts such as those in the *birthday* relation. Similarly we have corresponding alphabets of sorted *variables*. An identifier starting by the symbol $ implicitly denotes a variable. A *term* is a variable or a constant.

A *schema* is an expression (Π, $\mathcal{E}$, $\mathcal{I}$, σ) where Π is a (possibly infinite) set of peer IDs; $\mathcal{E}$ and $\mathcal{I}$ are disjoint sets, respectively, of *extensional* and *intensional* names of the form $m@p$ for some relation name m and some peer p; and the typing function σ defines for each $m@p$ in $\mathcal{E} \cup \mathcal{I}$ the arity and sorts of its components. Note that because $\mathcal{I} \cap \mathcal{E} = \varnothing$, no m is both intensional and extensional in the same p. Considering Π to be infinite reflects the assumption that the set of peers is dynamic and of unbounded size (we can discover or create new peers) just like it is the case on the Web.

Facts and rules

Given a relation $m@p$, a (ground) (p-)*fact* is an expression $m@p(\overline{u})$ where $\overline{u}$ is a vector of data elements of the proper types, i.e., correct arity and correct sort for each component. For a set K of facts and a peer p, $K[p]$ is the set of p-facts in K. The notion of fact is central to the model. It will be the basis for both stored knowledge and communication. For instance, in the peer p, if we derive the extensional fact $r@p(1,2)$, this is a fact p knows. On the other hand, if we derive the extensional fact $s@q(2,3)$, this is a message that p sends to q.

A *(*Webdamlog*) rule* is an expression of the form

$$M_{n+1}@Q_{n+1}(\overline{U}_{n+1}) \text{ :- } (\neg)M_1@Q_1(\overline{U}_1)...(\neg)M_n@Q_n(\overline{U}_n)$$

where each M_i is a relation term, each Q_i is a peer term and each $\overline{U}_i$ is a vector of data terms. We also allow in the body of the rules, atoms of the form $X = Y$ or $X \neq Y$ where X, Y are terms.

We require a rule to be *safe*, i.e.,

1. For each i, if Q_i is a peer variable, it must be previously bound, i.e., it must appear in $\overline{U}_j$ for some positive literal $M_j@Q_j(\overline{U}_j), j < i$.

2. Each variable occurring in a literal $\neg M_i@Q_i(\overline{U}_i)$ must be previously bound to a positive literal.

3. Each variable in the head must be positively bound in the body.

Remark 3.1 (Unguarded peer)**.** Observe that we treat differently peer and relation names. By (1), a peer variable has to be previously positively bound. We insist on (1) so that we control explicitly to whom a peer sends a message or delegates a rule.

Note also that because of (1), the ordering of literals is relevant. One could define a variation of the language, namely *peer-unguarded* Webdamlog by not imposing Constraint (1) and considering all orderings of the body literals (with the negative ones seen implicitly after all the others).

We say that a rule is *deductive* if the head relation is intensional. Otherwise, it is *active*. Rules live in peers. We say that a rule in a peer p is *local* if all Q_i in all body relations are from p. It is *fully local* if the head relation is also from p. We will see that the following four classes of rules play different roles:

Local deduction Fully local deductive rules are used to derive intensional facts *locally*.

Update Local active rules are used for sending messages, i.e., facts, that modify the extensional databases of each peers that receive them.

View delegation The local but not fully local deductive rules provide some form of view materialization. For instance, this rule results in providing at q a view of some data from p:

$$at\ p:\ r@q(\overline{U})\ \text{:-}\ (\neg)r_1@p(\overline{U_1}),...(\neg)r_n@p(\overline{U_n})$$

General delegation The remaining rules allow a peer to install arbitrary rules at other peers.

Peer and relation variables provide considerable flexibility for designing applications. However, observe that because of them, it may be unclear whether a rule is (fully) local or not, deductive or active. Note that in a real system, one can wait until a rule is (partially) instantiated at runtime to find what its nature is, and decide what should be done with it.

The semantics of *Webdamlog* is based on autonomous local computations of the peers. We consider this first, then look at the global semantics of *Webdamlog*.

Local computation

A local computation happens at a particular peer. Based on its set of facts and set of rules, the peer performs the following: (1) some local deduction of

intensional facts, (2) the derivation of extensional facts that either define its next state or are sent as messages, and (3) the delegation of rules to other peers.

(Local deduction) For local deduction, we want to rely on the semantics of standard Datalog languages. However, because of possible relation variables, *Webdamlog* rules are not strictly speaking proper Datalog$^\neg$ rules, since the relation names of atoms may include variables. So, to specify local deduction, we proceed as follows. We start by *grounding* the peer and relation variables appearing in the rules. More precisely, for each rule

$$M_{n+1}@Q_{n+1}(\overline{U}_{n+1}) \text{ :- } (\neg)M_1@Q_1(\overline{U}_1)...(\neg)M_n@Q_1(\overline{U}_n)$$

of peer p, we consider the set of rules obtained by instantiating relation variables M_i with relation constants and peer variables Q_i with peer constants. To ensure finiteness, we only use constants from the active domain of the peer, that is, that appear in some fact or rule in the peer state. We can now deal with pairs $m@p$ of relation and peer constants as normal relation symbols in Datalog. Since for local deduction, we are only interested in fully local deductive rules, we will remove rules with a relation $m@q$ for $q \neq p$ or an extensional relation in the head. We must also remove rules that violate the arity or sort constraints of σ. The remaining rules are all fully local deductive rules which belong to standard Datalog.

Now, given a set I of facts and a set P_d of fully local deductive rules (defined as in the previous paragraph), we denote by $P_d^*(I)$ the set of facts inferred from I using P_d with a standard Datalog semantics. For instance, in absence of negation, the semantics is, as in classical Datalog, the least model containing I and satisfying P_d. When considering negation, one can use any standard semantics of Datalog with negation, say well-founded [Prz90] or stable [GL88]. For results in Section 3.4.2, we will use a variant of stratified negation semantics [CH85]. So we assume the program is stratified with respect to negation.

(Updates) Given a set K of facts and a set P_a of local active rules, the set $P_a(K)$ of *active consequences* is the set of *extensional* facts $v(A)$ such that for some rule A :- Θ of P_a and some valuation v, $v(\Theta)$ holds in K, and $v(A), v(\Theta)$ obey the typing and sort constraints of σ. This is the set of *immediate consequences*. Note that it does not necessarily contain all facts in K.

Observe that for deductive rules, we typically use a fixpoint (based on the particular semantics that is used), whereas for active rules, we use the immediate consequence operator that is explicitly procedural.

(Delegation) Given a set K of facts and a set P of (active and deductive) rules in some peer p, the *delegation* $\gamma_{pq}(P, K)$ of peer p to $q \neq p$ is defined as follows.

If for some deductive rule $M@Q(\overline{U})$:- Θ in P, there exists a valuation v such that $v\Theta$ holds in K, $v(Q) = q$, and the typing constraints in σ are respected, then

$$vM@vQ(v\overline{U}) \text{ :-}$$

is in $\gamma_{pq}(P, K)$.

If for some active or deductive rule

$$A \text{ :- } \Theta_0, (\neg)M@Q(\overline{U}), \Theta_1$$

in P (where Θ_0, Θ_1 are sequences of possibly negated atoms), there exists a valuation v satisfying σ such that $v\Theta_0$ contains only p-facts, $v\Theta_0$ holds in K, and $vQ = q(\neq p)$, then

$$vA \text{ :- } (\neg)M@vQ(v\overline{U}), v\Theta_1$$

is in $\gamma_{pq}(P, K)$.

Nothing else appears in $\gamma_{pq}(P, K)$.

Observe that we do not produce facts that are improperly typed. In practice, a peer p may not have complete knowledge of the types of some peer q's relations. Then p may "derive" an improperly typed fact. This fact will be sent and rejected by q. From a formal viewpoint, it is simply assumed that the fact has not even been produced. Similarly, a peer may delegate an improperly typed rule, but that rule will never produce any facts, and so can safely be ignored.

We are now ready to specify the semantics of the *Webdamlog* language.

States and runs

A *(*Webdamlog*)* *state* of the schema $(\Pi, \mathcal{E}, \mathcal{I}, \sigma)$ is a triple $(I, \Gamma, \tilde{\Gamma})$ where for each $p \in \Pi$, $I(p)$ is a finite set of extensional p-facts at p, $\Gamma(p)$ is the finite set of rules at p, and $\tilde{\Gamma}(p, q)$ $(p \neq q)$ is the set of rules that p delegated to q. For $p \in \Pi$, the (p-)*move* from $(I, \Gamma, \tilde{\Gamma})$ to $(I', \Gamma', \tilde{\Gamma}')$ (corresponding to the *firing* of peer p) is defined as follows. Let P_p be $\Gamma(p) \cup (\cup_q \tilde{\Gamma}(q, p))$, P_{pd} be the set of fully local deductive rules in P_p and P_{pa} the set of local active rules in it. Then the next state is defined as follows:

- (Local deduction) Let $K = P^*_{pd}(I(p))$.

- (Updates) $I'(p) = P_{pa}(K)[p]$; and
  (external activation) $I'(q) = I(q) \cup P_{pa}(K)[q]$ for each $q \neq p$.
- (Delegations) $\tilde{\Gamma}'(p,q) = \gamma_{pq}(P_p, K)$ for each q ≠ p; and
  $\tilde{\Gamma}'(p',q') = \tilde{\Gamma}(p',q')$ otherwise.

A (*Webdamlog*) *system* is a state $(I, \Gamma, \tilde{\Gamma})$ where $\tilde{\Gamma}(p,q) = \varnothing$. We will speak of the system (I, Γ) (since $\tilde{\Gamma}$ is empty). A sequence of moves is *fair* if each peer p is invoked infinitely many times. A *run* of a system (I, Γ) is a fair sequence of moves starting from (I, Γ).

Observe that $I(p)$ is finite for each peer and that it remains so during a run, even if the number of peers is infinite. Note also that deletions are implicit: a fact is deleted if it is not derived for the next state. We recall that facts can be made persistent using persistence rules of the form

$$r@p(\overline{\mathrm{U}}) \text{ :- } r@p(\overline{\mathrm{U}}), \neg del.r@p(\overline{\mathrm{U}})$$

In the following, such a rule for relation $r@p$ will be denoted *persistent* $r@p$.

Remark 3.2 (Fact and rules)**.** It is important to observe a difference between the semantics of facts and rules. Observe that, if we visit twice peer p in a row, the fact-messages that p sends to q accumulate at q. On the other hand, the new set of delegations replaces the previous such set. Moreover, when we visit q, the messages of q are consumed whereas the delegations stay until they are replaced. These subtle differences are important to capture different facets of distributed computing, e.g., for capturing materialized views or for providing the expected semantics to extensional / intensional data.

3.2 Key observations

In this section, we present examples that illustrate the interest of our model for distributed data management, and make key observations about different aspects of the model.

We first consider two serious criticisms that could be adressed to the model, namely too much synchronization and too little local control. We show how both issues can be resolved.

Too much synchronization

Observe that moves capture some form of asynchronicity and parallelism. The peer that fires is randomly chosen and does (atomically) some processing. However, there is still some form of synchronization, that may be undesired.

When we process peer p, messages from p to some peer q are instantaneously available in q. This is impossible to guarantee in practice. In a standard manner, when a more precise modeling is desired, one can introduce a peer acting as the network between p and q. Instead of going instantaneously from p to q, the message goes instantaneously from p to $network_{pq}$, waits there until $network_{pq}$ is fired, then goes instantaneously to q, and similarly for delegations. This captures more realistically what happens in practice, and does not require changing the model.

Too little local control

In the model we have defined, nothing prevents a peer p from modifying another peer q's relations or accessing q's data using delegation. In realistic settings, one would want a peer to be able to hold private information, which cannot be modified or accessed by another peer without its permission. This can be easily accomplished by extending the model with *local relations*. These relations can only appear in p's own facts and rules (i.e., $I(p)$ and $\Gamma(p)$), but not in any rules delegated to p (in practice, this means p would simply ignore any delegations using one of its private relations).

To illustrate, suppose that we want to control the access to a relation $r@p$ of peer p. We create for this purpose two local relations $read@p(\$r, \$q)$ and $write@p(\$r, \$q)$ that store who can read/write in p's relations. Note that the *read* and *write* relations are local, i.e., only p can specify the access rights in p. Relations $r@p$ and $del.r@p$ must also be local so that p control access to them. To obtain relation $r@p$, a peer q sends a message $get@p(r, q)$. The following rule controls whether q will receive the data it requested:

```
at p: send@$q($r,$x) :- get@p($r,$q), read@p($r,$q),
                        $r@p($x)
```

Insertions in $r@p$ (or deletions using $del.r@p$) are treated similarly. Access control in *Webdamlog* is at the center of an on-going work in [1].

We next consider two subtleties of delegation.

Delegation and complexity

Consider the rule:

at p: m@q() :- m_1@p($q,$x), m_2@$q($x)

If there are 1000 distinct tuples $(p_i, 0)$ such that $m_1@p(p_i, 0)$ holds, then we have to install rules in 1000 distinct peers. So delegation is inherently transforming data complexity into program complexity.

Peer life and delegation

It is very simple in the model to consider that peers are born, die or hibernate. We simply have to insist that p can be fired (p-move) only if p is alive and not hibernating. We can assume that messages and delegations to dead peers are simply lost and that for hibernating ones, they are buffered somewhere in the network. A subtlety is that (with this variant of the model), if a peer dies without cleanly terminating, delegations from this peer are still valid. In practice, the system may realize that a particular peer is no longer present and terminate its delegations.

We conclude this section with three examples that illustrate different aspects of the language, communications, persistence services, and rule updates.

Multicasting

We can simulate channels, i.e., m-n communications with the following rules:

```
at q: persistent channelsubscribe@q
      channel@$p($m,q,$s) :- channelsubscribe@q($p,$m),
          $m@q($s)
```

The rules at peer q allows him to support channels. A peer p can subscribe to receiving all the messages from the channel m hosted by q by sending: $channelsubscribe@q(p, m)$ to q. Then, whenever someone sends a message $m@q(s)$, p will receive $channel@p(m, q, s)$.

Database server replication

The following rule allows a database server to replicate relations from many peers:

```
intensional export@db(relation,peer)
at db: persistent tobeexported@db
       export@db($r,$p,$x) :- tobeexported@db($r, $p),
           $r@$p($x)
```

If a peer p wants his relation $r@p$ to be stored at db, then p simply needs to send db the message $tobeexported@db(r, p)$. Now, $export@db(r, p, \$x)$ is a copy of $r@p(\$x)$.

Rule updates and rule deployment

Observe that (to simplify) we assumed that the set of rules in a run is fixed, i.e., $\Gamma(p)$ is fixed for each p. It is straightforward to extend the model to support addition or deletion of rules. Furthermore, one might want to be able to control whether a particular rule is deployed on a particular peer. To illustrate this point, consider the two rules:

at p: persistent server@p
f@$p($u) :- server@p($p), f_1@$p(u_1$),...,f_n@$p(u_n$)

Sending the message $server@p(q)$ results in installing

at q: f@q($u) :- f_1@q(u_1$),...,f_n@q(u_n$)

Note that if we send the message $del.server@p(q)$, the rule is removed.

3.3 Expressive power

In this section, we study the expressive power of *Webdamlog* and of different languages that are obtained by allowing or restricting delegations. We also consider the expressive power of timestamps. More precisely, we consider the following languages for rules:

- WL (*Webdamlog*): the general language.
- VWL (views WL): the language obtained by restricting delegations to only view delegations.
- SWL (simple WL): the language obtained by disallowing all kinds of delegations.

At the core of view delegation, we find the maintenance of materialized views. To maintain views, we will see that timestamps turn out to be useful. More precisely, for time, we assume that each peer has a local predicate called *time* (with $time(t)$ specifying that the current move started at local time t). The predicate $<$ is used to compare timestamps. Note that each peer has its separate clock, so the comparison of timestamps of distinct peers is meaningless. To prevent time from being a source of nondeterminism, for t_1, t_2 two times at different peers, we assume: $t_1 \not< t_2$ and $t_2 \not< t_1$ (the time from two peers are incomparable). The languages obtained by extending the previous languages with timestamps are denoted as follows: WL^t, VWL^t, SWL^t.

3.3.1 Traces and simulations

To formally compare the expressivity of the above languages, we need to introduce the auxiliary notions of trace and simulation.

Let $r = (I_1, \Gamma_1, \widetilde{\Gamma}_1), ...(I_n, \Gamma_n, \tilde{\Gamma}_n), ...$ be a run. Let M be a set of predicates and I a set of facts. Then $\Pi_M(I)$ is the set of facts in I with predicates in M. The *M-trace* of the run r for a set M of predicates is the subsequence of $\pi_M(I_{i_1}), ..., \pi_M(I_{i_n})...$ obtained by starting from $\pi_M(I_1), ..., \pi_M(I_n)...$ and removing all repetitions, i.e., deleting the $(k+1)$th element of the sequence if it is identical to the kth, until the sequence does not contain two identical consecutive elements. Given an initial state S and a set of predicates M, we denote by *M-trace*(S) the set of M-traces of runs from S. In some sense, it is what can be observed from S when only facts over M are visible.

Let α be a set of peers. An initial state $S = (I, \Gamma)$ can be *α-simulated* by an initial state $S' = (I, \Gamma')$ if $\Gamma(p) = \Gamma'(p)$ for all $p \in \alpha$ and S and S' have the same M-traces, where M is the set of relations of S. In other words, from the point of view of what is visible from S, S' behaves exactly like S. The set of peers α is meant to capture the part of the system (one or more peers) that we want to keep strictly identical.

Now, we say that a language L can be *simulated* by a language L', denoted $L \prec L'$, if there exists a translation τ from programs in L to programs in L' such that for each initial state (I, Γ) (with programs in L) and for each α, $(I, \overline{\tau}(\Gamma))$ α-simulates (I, Γ) where $\overline{\tau}$ is defined by: for each peer p,

- if $p \in \alpha$, $\overline{\tau}(\Gamma(p)) = \Gamma(p)$.
- otherwise, $\overline{\tau}(\Gamma(p)) = \tau(\Gamma(p))$.

Clearly, in the previous definition, the peers in α are not part of the simulation, they behave exactly as originally. In some sense, they should not even be aware that something has changed.

3.3.2 Expressivity results

The expressive power of the different languages are compared in Figure 3.1. The containments are strict except for that of VWL^t inside WL^t where the issue remains open.

Our first result states that view delegation cannot be simulated by simple rules.

Theorem 3.3 (No views in SWL)**.** *VWL $\not\prec$ SWL.*

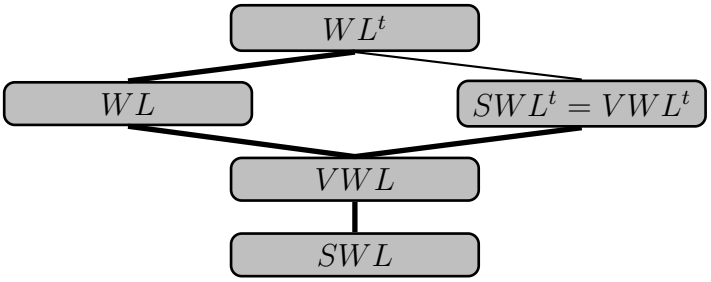


Figure 3.1: Expressive power of the rule languages (the inclusion is strict when the arc is in bold)

Proof. Intuitively, the difficulty is that the system may visit an arbitrary number of times the same peer p before visiting another peer q. Then q sees all the messages from p at the same time and ignores in which order they were received.

Formally, consider a VWL system (I, Γ) consisting of three peers p_α, p, q. There are two facts that hold in the initial state: $true@p_\alpha()$, $true@p()$.

The set of active rules $\Gamma(p_\alpha)$ maintain the peer p_α in a permanent flip-flop between two modes:

$$
\begin{array}{l}
at\ p_\alpha : r@p() \text{ :- } true@p_\alpha() \\
\qquad\quad false@p_\alpha() \text{ :- } true@p_\alpha() \\
\qquad\quad del.r@p() \text{ :- } false@p_\alpha() \\
\qquad\quad true@p_\alpha() \text{ :- } false@p_\alpha()
\end{array}
$$

Note that p_α keeps inserting then deleting the same proposition in p, namely $r@p()$. Peer p uses the following four rules:

$$
\begin{array}{l}
at\ p : r@p() \text{ :- } r@p(), \neg del.r@p() \\
\qquad\quad true@p() \text{ :- } false@p() \\
\qquad\quad false@p() \text{ :- } true@p() \\
\qquad\quad s@q() \text{ :- } r@p()
\end{array}
$$

The first active rule maintains relation $r@p$. The next two active rules maintain p in a flip-flop between two modes. The last rule is a view delegation rule. It is because of this latter rule that the system is in VWL but not in SWL.

Finally peer q has one active rule:

$$at\ q : true@q() \text{ :- } s@q()$$

Suppose for a contradiction that there is a p_α-simulation of this system in SWL, via some program translation function τ. As the set of peers is finite

(namely 3), the initial state $(I, \tau(\Gamma))$ is finite. Thus, it includes a finite set of relation names and constants. This means that there is a finite number of distinct messages that can be sent during a run of this system. Now let r_1 be any run of $(I, \tau(\Gamma))$ such that the initial segment of activated peers is as follows: p_α, then p, then p_α, then p, etc., n times (for n to be fixed later in the proof), and then q. Let $I, I_1, I_2, ..., I_{2n-1}, I_{2n}, I'$ be the trace of r_1. Because of the two flip-flops, the trace has this size and it is clear from it which peer has been activated at each step.

Consider a second run r_2 which is defined like r_1 except that this time we visit p_α and p, $n+1$ times, then q. Let $I, I_2, I_3..., I_{2n-1}, I_{2n}, I_{2n+1}, I_{2n+2}, I''$ be the trace of r_2.

Observe that while p and p_α are being activated, q is simply accumulating messages. Recall that the set of messages that q may accumulate is finite. Thus we can choose n large enough so that $I_{2n+2}(q) = I_{2n}(q)$. Suppose that $I'(q)$ contains $true@q$. Then because the set of messages at q is the same in the second run, $I''(q)$ also contains $true@q$, a contradiction because the last iteration in p_α, p must have removed $r@p$. A similar contradiction occurs if $true@q$ is not produced. Thus such a simulation does not exist. □

Next we separate VWL and WL.

Theorem 3.4 (No general delegations in VWL)**.**
WL $\not\leq$ VWL.

Proof. (sketch) Intuitively, peer q will use a general delegation to ask peer p to do something that is beyond the capability of the rules in p. This is not trivial because p may perform very complex operations with arbitrarily many complex rules. However, it turns out that there is a limit to what p can do. To prove it, we use the fact that with formulas using a bounded number k of variables, one cannot check whether a graph has a clique of size $k+1$ (when an ordering of the nodes is not available).

Formally, consider a WL system (I, Γ) that consists of three peers p_α, p, q. Intuitively, peer p_α sends a sequence of updates to a graph that is originally empty and is stored at p. To do that, p_α has a persistent relation that stores a sequence of updates. More precisely, p_α has a set of tuples of the form: $upd@p_\alpha(i, o, a, b)$ where i in [0,m] for some m and there is a single tuple for each i, o in { ins, del }, and a, b are data elements in a very large fixed set Σ (the identifiers of the graph g.) Peer p_α also has a persistent relation *next* containing the tuples: $[0,1], ...[m-1, m]$. Finally, p_α has the fact $now@p_\alpha(0)$ in its initial state. The program of p_α consists of the following active rules:

$at\ p_\alpha$: $g@p(\$x, \$y)$:- $now@p_\alpha(\$i), upd@p_\alpha(\$i, ins, \$x, \$y)$
$del.g@p(\$x, \$y)$:- $now@p_\alpha(\$i), upd@p_\alpha(\$i, del, \$x, \$y)$

$$now@p_\alpha(\$j) \text{ :- } now@p_\alpha(\$i), next(\$i, \$j)$$

Now p has the following active rule for maintaining the graph g :

$$at\ p : g@p(\$x, \$y) \text{ :- } g@p(\$x, \$y), \neg del.g@p(\$x, \$y)$$

Finally, peer q has a rule delegation to p:

$$at\ q : clique@q() \text{ :- } \wedge_{1 \leqslant i,j \leqslant n} g@p(\$x_i, \$x_j), \$x_i \neq \$x_j$$

which essentially requests p to send a message if there exists an n-clique in $g@p$. Peer q also has a flip-flop rule:

$$\begin{aligned} at\ q : true@q() &\text{ :- } false@q() \\ false@q() &\text{ :- } true@q() \end{aligned}$$

Originally $true@q()$ holds.

Suppose for a contradiction that there is a p_α-simulation of this system in *VWL*. Consider the run of (I, Γ) beginning with a very long sequence $q(p_\alpha)^* p(p_\alpha)^* ... p$ where each time p is called, the graph oscillates between "there is a clique" and "there isn't". Note that the first time q is called, it installs the delegation.

Let k be the number of variables and constants that appear in a rule in $\tau(\Gamma(p))$. As the rules in p have less than k symbols, they can only evaluate formulas in FO^k. Choose $n > k$, so that formulas in FO^k cannot check for the presence of an n-clique in a graph. Choose also the set of node identifiers Σ large enough. (Recall that the translation for the rules of p is independent from the program of q and p_α.) So, it is not possible for p to evaluate whether there is a clique. So q has to be called before each *clique* message to check the existence of a clique. Note that it is possible to do so: p pretends it has not been called and waits until q is called; then q sends a secret message to p to tell p whether there is a clique.

This is "almost" a simulation except that q has a bounded memory that depends essentially on Σ. Now consider a very long sequence of the WL system that never calls q. If the sequence is long enough, its simulation in VWL will visit twice the same state. Then by pumping, one can construct an infinite run of the VWL simulating system such that the flip-flop of q is never activated. This corresponds to a simulation of an unfair run of the WL system, a contradiction. Thus there can be no VWL simulation of the above WL system. □

We now consider timestamps. The next result compares the expressive power of WL and WL^t.

Theorem 3.5 (Timestamps)**.** *For a finite number of peers,*

1. *WL is in* PSPACE*;*
2. *SWL^t over a single peer can simulate any arbitrary Turing machine;*
3. *Thus, SWL^t $\not\leftarrow$ WL and (a fortiori) WL^t $\not\leftarrow$ WL.*

Proof. (sketch) For (1.), consider a fixed schema over a finite number of peers. Let (I, Γ) be an initial instance of size $n = |I| + |\Gamma|$. Let $(I_i, \Gamma, \widetilde{\Gamma}_i)$ be an instance that is reached during the computation. Because the schema is fixed, the number of facts that can be derived is bounded by a polynomial in n, and each fact is also of bounded size. So, $|I_i|$ can be bounded by a polynomial in n. Similarly, the size of $\widetilde{\Gamma}_i$ can be bounded by a polynomial in n, since a rule that is delegated is essentially determined by an instantiation of an original rule and a position in it. Thus we can represent $(I_i, \Gamma, \widetilde{\Gamma}_i)$ in polynomial space in n. Hence, WL is in PSPACE.

Now consider (2.). Let M be a Turing Machine. We can assume without loss of generality that it is deterministic and that it has a tape that is infinite only in one direction. The SWL^t system that simulates it is as follows. Its initial instance encodes the initial state of M. More precisely, it has a relation *input*, with initial value

$$\{\ \mathit{input}(0,1,a_1),\ \mathit{input}(1,2,a_2),\ \ldots\ \mathit{input}(n{-}1,n,a_n)\ \}$$

where $a_1a_2...a_n$ is the input of M. It also has a relation *tape* that is originally empty.

First, the SWL^t system copies the input on its tape using the timestamps $t_0, t_1, t_2...$ to identify tape cells. More precisely, it constructs,

$$\{\mathit{tape}(t_0,t_1,a_1,s_0),\mathit{tape}(t_1,t_2,a_2,\bot),...,\mathit{tape}(t_{n-1},t_n,a_n,\bot)\}$$

where s_0 is the start state of M. Using rules from SWL^t, it is straightforward to simulate moves of M. The only subtlety is that at each step of the iteration, the tape is augmented so that there is no risk of reaching its limit. The fact that the cells are denoted with timestamps guarantees that no two cells will have the same ID.

Now, given the encoding of a word w, one can simulate the computation of TM on w. Thus (2), so (3). □

Note that the converse of (1) holds: any PSPACE query over an ordered database can be computed in SWL (hence WL) with a single peer. This can be shown by proving how to simulate in SWL with a single peer, the language Datalog$^{\neg\neg}$ that can express all PSPACE queries on ordered databases [AV91].

Next we see how to use timestamps to simulate view maintenance.

Theorem 3.6 (Views with timestamps)**.** *VWL^t $\approx$ SWL^t.*

Proof. (sketch) We illustrate with an example the simulation of view delegation by a program with timestamps.

Consider a VWL system with an extensional relation $s@q$ and the deductive rule at p: $r@p(\overline{U})$:- $s@q(\overline{U})$ that specifies that $r@p$ is a view of $s@q$. The simulation of the view delegation in SWL^t is as follows.

at q : *persistent* $past@q$
$aux@p(\overline{U}, \$t)$:- $s@q(\overline{U})$, $time@q(\$t)$
$past@q(\$t)$:- $time@q(\$t)$
$obsolete@p(\$t)$:- $past@q(\$t)$

at p : *intensional* $r@p$
persistent $aux@p$, $obsolete@p$
$r@p(\overline{U})$:- $aux@p(\overline{U}, \$t)$, $\neg$ $obsolete@p(\$t)$

Then the value of $r@p$ is that of $s@q$ when q was last visited, i.e., $r@p$ is a copy of $s@q$ at the last visit of q.

The above simulation is straightforwardly generalized to arbitrary VWL systems, from which we obtain the desired $\text{VWL}^t \approx \text{SWL}^t$. □

It is still open whether $\text{WL}^t \not\prec \text{VWL}^t$.

3.4 Convergence of *Webdamlog*

Systems that converge to a unique state independently of the order of computation, i.e., some form of Church-Rosser property, are of particular interest. In this section, we consider two kinds of such systems: the positive and the strongly-stratified *Webdamlog* systems. Indeed, we show that such systems continue to converge even in presence of insertions of facts or rules. Finally, we show that for these two classes of systems, the distributed semantics can be seen as mimicking the centralized semantics.

3.4.1 Positive *Webdamlog*

Clearly, negation may explain why a system does not converge. However, the following example shows that even in absence of negation, convergence is not guaranteed because the order of arrival of messages matters:

Example 3.7. Consider the rules:

```
at q:  extensional r1@q, r2@q, r@q
       persistent r@q
       r@q() :- r1@q(), r2@q()
```

at q1: r1@q() :-
at q2: r2@q() :-

If we process the peers according to the order $q1, q, q2, q, q1, \ldots$, then $r@q$ is never derived. If we consider instead the order $q1, q2, q, q1, q2, q, \ldots$, then $r@q$ is derived and remains forever. The absence of convergence here is in fact a desired feature of the model: the extensional relations model events, so their arrival times matter.

On the other hand, note that, as we will see, if in the example $r1@q$ and $r2@q$ were intensional, the system would converge.

We now introduce the restricted systems we study in this section. A *Webdamlog* state or system is *positive* if the following holds:

1. Each of its rules is positive (no negation); and

2. Each extensional relation $m@p$ is made persistent with a rule of the form $m@p(\overline{U})$:- $m@p(\overline{U})$.

We will see that because of these restrictions, the states in runs of positive systems are monotonically increasing. For positive systems with a finite number of peers, there are only finitely many possible states, so monotonicity implies that runs converge after a finite number of steps. We will also show convergence for positive systems with infinitely many peers, except that in this case, we may converge only in the limit. This motivates the following somewhat complex definition of convergence.

A run $S_0, S_1, S_2, \ldots$ *converges* to a possibly infinite state $S^* = (I^*, \Gamma^*, \widetilde{\Gamma}^*)$ if for each finite $S' \subseteq S^*$, there exists $k_{S'}$ such that for all $k > k_{S'}$, $S' \subseteq S_k$ and if for each finite $S' \not\subseteq S^*$, there is $k_{S'}$ such as for all $k > k_{S'}$, $S' \not\subseteq S_k$. We say a system S *converges* if all its runs converge to the same state.

The following theorem states the convergence of (possibly infinite) positive systems.

Theorem 3.8 (Convergence)**.** *All positive* Webdamlog *systems converge.*

Lemma 3.9. *Suppose* $I_1(p^*) \subseteq I_2(p^*)$, $\Gamma_1(p^*) = \Gamma_2(p^*)$ *and* $\widetilde{\Gamma}_1(q, p^*) \subseteq \widetilde{\Gamma}_2(q, p^*) \forall q \neq p^*$. *Let* $P_{a,i}$ *(resp.* $P_{d,i}$*) be the set of local active (resp. fully local deductive) rules in* $\Gamma_i(p^*) \cup \cup_{q \neq p^*} \widetilde{\Gamma}_i(q, p^*)$. *Then if there is no negation in the rules, we have* $P_{a,1}(K_1) \subseteq P_{a,2}(K_2)$ *and*

$$\gamma_1(p^*, q)(P_{a,1}, K_1) \subseteq \gamma_2(p^*, q)(P_{a,2}, K_2) \forall q \neq p^*$$

where $K_i = P^*_{d,i}(I_i(p^*)))$.

Proof. (of Lemma 3.9) Since $\Gamma_1(p^*) = \Gamma_2(p^*)$ and $\widetilde{\Gamma}_1(q, p^*) \subseteq \widetilde{\Gamma}_2(q, p^*)$ for all $q \neq p^*$, it follows that $P_{a,1} \subseteq P_{a,2}$ and $P_{d,1} \subseteq P_{d,2}$. Together with $I_1(p^*) \subseteq I_2(p^*)$, and in absence of negation, we obtain $P_{a,1}(P^*_{d,1}(I_1(p^*))) \subseteq P_{a,2}(P^*_{d,2}(I_2(p^*)))$. Likewise, $\gamma_{p^*q}(P_{a,1}, P^*_{d,1}(I_1(p^*))) \subseteq \gamma_{p^*q}(P_{a,2}, P^*_{d,2}(I_2(p^*)))$. □

Proof. (of Theorem 3.8) In fact, we will prove that the result is true for a simple update I', Γ', since the result is then easy to generalize. Consider a positive *Webdamlog* system $(I_0, \Gamma_0, \widetilde{\Gamma}_0)$. Let $r = (I_0, \Gamma_0, \widetilde{\Gamma}_0)(I_1, \Gamma_1, \widetilde{\Gamma}_1)$ $(I_2, \Gamma_2, \widetilde{\Gamma}_2) \ldots$ be a run for this system. It follows from the definition of moves that $\Gamma_i = \Gamma_j$ for all $i, j \geqslant 0$ and that delegated rules are sub-rules of these sets so have no negation. So $(I_i, \Gamma_i, \widetilde{\Gamma}_i)$ is positive for every $i \geqslant 0$. We show by induction on i that $I_i(p) \subseteq I_{i+1}(p)$ and $\widetilde{\Gamma}_i(p, q) \subseteq \widetilde{\Gamma}_{i+1}(p, q)$ for all i and all peers p, q, i.e., the states in the run increase monotonically. Using this property, it is easy to show that r converges to the (possibly infinite) state $(I^*, \Gamma_0, \widetilde{\Gamma}^*)$ where $I^*(p) = \cup_i I_i(p)$ and $\widetilde{\Gamma}^*(p, q) = \cup_i \widetilde{\Gamma}_i(p, q)$. The base case $(i = 0)$ for our induction is straightforward. If the first move is a p^*-move, then by the definition of move, we have $I_0(q) \subseteq I_1(q)$ for all $q \neq p^*$. For peer p^*, we use the fact that $I_0(p)$ contains only extensional p-facts and that $\Gamma_0(p)$ contains persistence rules for all extensional relations of p. We thus obtain $I_0(p^*) \subseteq I_1(p^*)$. As for delegations, we have $\widetilde{\Gamma}_0(p, q) = \varnothing$ for all p, q (since $(I_0, \Gamma_0, \widetilde{\Gamma}_0)$ is initial), hence $\widetilde{\Gamma}_0(p, q) \subseteq \widetilde{\Gamma}_1(p, q)$ for all peers p, q. Suppose next that the claim holds for all $i < k$. Let p^* be the peer whose move takes $(I_k, \Gamma_k, \widetilde{\Gamma}_k)$ to $(I_{k+1}, \Gamma_{k+1}, \widetilde{\Gamma}_{k+1})$. Using the same argument as in the base case, we obtain $I_k(p) \subseteq I_{k+1}(p)$ for all peers p. According to the definition of moves, $\widetilde{\Gamma}_k(p, q) = \widetilde{\Gamma}_{k+1}(p, q)$ whenever $p \neq p^*$. Thus, the only interesting case is when $p = p^*$ and $\widetilde{\Gamma}_k(p^*, q) \neq \varnothing$. In this case, we must have visited peer p^* previously. Let j be such that the last p^*-move took $(I_j, \Gamma_j, \widetilde{\Gamma}_j)$ to $(I_{j+1}, \Gamma_{j+1}, \widetilde{\Gamma}_{j+1})$. Since our last visit to p^* was at timepoint j, $\widetilde{\Gamma}_{j+1}(p^*, q) = \widetilde{\Gamma}_k(p^*, q)$. By repeatedly applying the IH, we obtain $I_j(p) \subseteq I_k(p)$ and $\widetilde{\Gamma}_j(p, q) \subseteq \widetilde{\Gamma}_k(p, q)$ for all peers p, q. In particular, we have $I_j(p^*) \subseteq I_k(p^*)$, $\Gamma_j(p^*) = \Gamma_k(p^*)$, and $\widetilde{\Gamma}_j(p^*, q) \subseteq \widetilde{\Gamma}_k(p^*, q)$. Applying Lemma 3.9, we get $\widetilde{\Gamma}_{j+1}(p^*, q) \subseteq \widetilde{\Gamma}_{k+1}(p^*, q)$, which yields the desired $\widetilde{\Gamma}_k(p^*, q) \subseteq \widetilde{\Gamma}_{k+1}(p^*, q)$, and completes our proof of the monotonicity claim.

Now consider two runs $r_1 = (I_{0,1}, \Gamma_{0,1}, \widetilde{\Gamma}_{0,1})(I_{1,1}, \Gamma_{1,1}, \widetilde{\Gamma}_{1,1})(I_{2,1}, \Gamma_{2,1}, \widetilde{\Gamma}_{2,1}) \ldots$ and $r_2 = (I_{0,2}, \Gamma_{0,2}, \widetilde{\Gamma}_{0,2})(I_{1,2}, \Gamma_{1,2}, \widetilde{\Gamma}_{1,2})$ $(I_{2,2}, \Gamma_{2,2}, \widetilde{\Gamma}_{2,2}) \ldots$ for the system which converge respectively to $(I^*_1, \Gamma^*_1, \widetilde{\Gamma}^*_1)$ and $(I^*_2, \Gamma^*_2, \widetilde{\Gamma}^*_2)$. We will prove by induction on $i \geqslant 0$ that for every state $(I_{i,1}, \Gamma_{i,1}, \widetilde{\Gamma}_{i,1})$ of r_1, there is $j \geqslant 0$ such that $I_{i,1}(p) \subseteq I_{j,2}(p)$ and $\widetilde{\Gamma}_{i,1}(p, q) \subseteq \widetilde{\Gamma}_{j,2}(p, q)$ for all peers p, q. This, together with monotonicity property in the previous paragraph, yields the

desired $(I_1^*, \Gamma_1^*, \widetilde{\Gamma}_1^*) = (I_2^*, \Gamma_2^*, \widetilde{\Gamma}_2^*)$. The base case ($i = 0$) is trivial since $(I_{0,1}, \Gamma_{0,1}, \widetilde{\Gamma}_{0,1}) = (I_{0,2}, \Gamma_{0,2}, \widetilde{\Gamma}_{0,2})$ (as they are both runs for the same system). For the induction step, suppose the claim holds for $i \leqslant k$, and consider $(I_{k+1,1}, \Gamma_{k+1,1}, \widetilde{\Gamma}_{k+1,1})$. Let p^* be the peer whose move takes $(I_{k,1}, \Gamma_{k,1}, \widetilde{\Gamma}_{k,1})$ to $(I_{k+1,1}, \Gamma_{k+1,1}, \widetilde{\Gamma}_{k+1,1})$. By the IH, we can find j such that $I_{k,1}(p) \subseteq I_{j,2}(p)$ and $\widetilde{\Gamma}_{k,1}(p,q) \subseteq \widetilde{\Gamma}_{j,2}(p,q)$ for all p, q. As r_2 is a fair run, we can find $l \geqslant j$ such as $(I_{l+1,2}, \Gamma_{l+1,2})$ results from a p^*-move. Since states are monotonically increasing in r_2, $I_{k,1}(p) \subseteq I_{j,2}(p) \subseteq I_{l,2}(p)$ and $\widetilde{\Gamma}_{k,1}(p,q) \subseteq \widetilde{\Gamma}_{j,2}(p,q) \subseteq \widetilde{\Gamma}_{l,2}(p,q)$ for all p, q. Using Lemma 3.9, $I_{k+1,1}(p^*) \subseteq I_{l+1,2}(p)$ and $\widetilde{\Gamma}_{k+1,1}(p,q) \subseteq \widetilde{\Gamma}_{l+1,2}(p,q)$ for all peers p, q. □

The previous theorem is still true if one allows the peers to insert facts and rules. One can show that the system will reach a stable state that does not depend on the points of insertion.

Theorem 3.10 (Updates)**.** *Given two positive* Webdamlog *systems (I,Γ) and (I′,Γ′), for any run of the system (I,Γ), if for a given step, I′ is added to the current set of facts and Γ′ to the current set of rules, then the modified run converges to the convergence state of (I ∪ I′,Γ ∪ Γ′).*

Proof. Let $(I_{0,1}, \Gamma_{0,1}, \widetilde{\Gamma}_{0,1}), (I_{1,1}, \Gamma_{1,1}, \widetilde{\Gamma}_{1,1})...$ be a run of (I,Γ); k a point of insertion; $(I_{k,1'}, \Gamma_{k,1'}, \widetilde{\Gamma}_{k,1'})$ the state $(I_{k,1} \cup I', \Gamma_{k,1} \cup \Gamma', \widetilde{\Gamma}_{k,1})$; and $r_1 = (I_{0,1}, \Gamma_{0,1}, \widetilde{\Gamma}_{0,1}), (I_{1,1}, \Gamma_{1,1}, \widetilde{\Gamma}_{1,1})...(I_{k-1,1}, \Gamma_{k-1,1}, \widetilde{\Gamma}_{k-1,1}), (I_{k,1'}, \Gamma_{k,1'}, \widetilde{\Gamma}_{k,1'}), (I_{k+1,1'}, \Gamma_{k+1,1'}, \widetilde{\Gamma}_{k+1,1'})...$ the modified run of the system. For ease of reference, we will denote by $(I_{i,1'}, \Gamma_{i,1'}, \widetilde{\Gamma}_{i,1'})$ any state $i \geqslant 0$ of this run. We show (i) that there is a run $r_2 = (I_{0,2}, \Gamma_{0,2}, \widetilde{\Gamma}_{0,2}), (I_{1,2}, \Gamma_{1,2}, \widetilde{\Gamma}_{1,2})...$ of the system $(I \cup I', \Gamma \cup \Gamma')$ such that for each $i \geqslant 0$, $I_{i,1'} \subseteq I_{i,2}$, $\Gamma_{i,1'} \subseteq \Gamma_{i,2}$ and $\widetilde{\Gamma}_{i,1'} \subseteq \widetilde{\Gamma}_{i,2}$, and (ii) that there is a run $r_3 = (I_{0,3}, \Gamma_{0,3}, \widetilde{\Gamma}_{0,3}), (I_{1,3}, \Gamma_{1,3}, \widetilde{\Gamma}_{1,3})...$ of the system $(I \cup I', \Gamma \cup \Gamma')$ such that for each $i \geqslant 0$, $I_{i,3} \subseteq I_{i+k,1'}$, $\Gamma_{i,3} \subseteq \Gamma_{i+k,1'}$ and $\widetilde{\Gamma}_{i,3} \subseteq \widetilde{\Gamma}_{i+k,1'}$. This is sufficient to prove the result since r_2 and r_3 are both runs of the same positive system, and thus must converge (by Theorem 3.8) to the same state. Since the states of r_1 are sandwiched between those of r_2 and r_3, convergence of both r_2 and r_3 to a single state implies convergence of r_1 to this same state.

Let us consider the first assertion. We select a run of the system $(I \cup I', \Gamma \cup \Gamma')$ with exactly the same sequence of peers as the modified run r_1. For $i = 0$, the desired inclusions clearly hold. Now suppose $i > 0$. Suppose $I_{i-1,1'} \subseteq I_{i-1,2}$, $\Gamma_{i-1,1'} \subseteq \Gamma_{i-1,2}$ and $\widetilde{\Gamma}_{i-1,1'} \subseteq \widetilde{\Gamma}_{i-1,2}$. Using Lemma 3.9, if $i \neq k$, we have the desired inclusions for timepoint i. If $i = k$, we have, using Lemma 3.9, $I_{k,1} \subseteq I_{k,2}$, $\Gamma_{k,1} \subseteq \Gamma_{k,2}$ and $\widetilde{\Gamma}_{k,1} \subseteq \widetilde{\Gamma}_{k,2}$. Since $I' \subseteq I_{0,2}$ and $\Gamma' \subseteq \Gamma_{0,2}$, and since the run of (I',Γ') is monotonic (by Theorem 3.8),

$I' \subseteq I_{k,2}$ and $\Gamma' \subseteq \Gamma_{k,2}$. Finally, since $I_{k,1'} = I_{k,1} \cup I'$, $\Gamma_{k,1'} = \Gamma_{k,1} \cup \Gamma'$ and $\widetilde{\Gamma}_{k,1'} = \widetilde{\Gamma}_{k,1}$, we have the result for $i = k$.

Now consider the second assertion. We choose a run r_3 of the system $(I \cup I', \Gamma \cup \Gamma')$ with exactly the same sequence of peers as the sub-run r_1 started from the timepoint k, i.e., if peer p moves at timepoint $i + k$ in r_1, then it is p who moves at timepoint i in r_3. It is clear that desired inclusions hold for $i = 0$, since the runs of (I,Γ) are monotonic. Let $i > 0$. Suppose $I_{i-1,3} \subseteq I_{i+k-1,1'}$, $\Gamma_{i-1,3} \subseteq \Gamma_{i+k-1,1'}$ and $\widetilde{\Gamma}_{i-1,3} \subseteq \widetilde{\Gamma}_{i+k-1,1'}$. Using Lemma 3.9, we obtain directly the desired inclusions for i. □

The previous theorem is straightforwardly extended to a series of updates. However, as illustrated by the following example, a more liberal definition of updates which also allows deletion of facts or rules in a system would compromise convergence.

Example 3.11. Consider the system defined as follows:

```
at p: extensional@p, intensional r@p
      r@q() :- r@p()
      r@p() :- s@p()
      s@p() :- s@p()
      s@p().
at q: intensional r@q
      r@p() :- r@q()
```

This system converges to a state where $I^*(p) = \{s@p()\}$, $\widetilde{\Gamma}^*(p,q) = \{r@q()\text{:-}\}$, $\widetilde{\Gamma}^*(q,p) = \{r@p()\text{:-}\}$ Then removing the fact $s@p()$ or the rule $r@p()$:- $s@p()$ after the convergence will not change $\widetilde{\Gamma}$ whereas $\widetilde{\Gamma}$ would be empty were the fact or the rule removed before beginning a run.

The previous example illustrates the difficulty of managing non-monotony. If we remove a fact or a rule, we need to remove as well all facts or rules that were deduced using this fact. This could be achieved using view maintenance techniques. We leave this for future work.

To further ground our semantics, we show that for positive systems, our semantics correspond to the standard centralized Datalog semantics.

Centralized semantics

In the positive case, we can compare with a "centralized" semantics, in which all facts and rules are combined into a single Datalog program. Such a comparison would not make sense in the general case since our semantics too closely depends on the order in which peers fire.

We associate to a positive *Webdamlog* state (I, Γ) the set $\cup_p(I(p) \cup \Gamma(p))$ composed of the facts and rules of all peers. We can transform this set of facts and rules into a standard Datalog program by first instantiating the variable relations in the rules (as was done for local computation) and then removing those rules that violate the typing constraints in σ. We denote by $c(I, \Gamma)$ the Datalog program thus obtained.

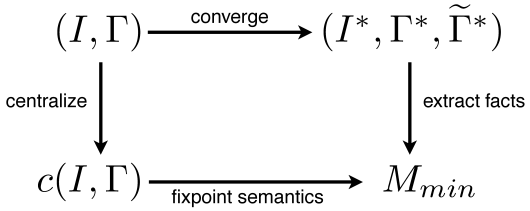


Figure 3.2: Link with centralized semantics

The following theorem (illustrated by Figure 3.2) demonstrates the equivalence, for the class of positive systems, of our distributed semantics and the traditional fixpoint semantics of Datalog. The result deals only with systems with finitely many peers to avoid having to extend Datalog to infinitely many relations.

Theorem 3.12. *Let (I, Γ) be a positive system with a finite number of peers that converges to $(I^*, \Gamma^*, \widetilde{\Gamma}^*)$, and let M_{min} be the unique minimal model of the Datalog program $c(I, \Gamma)$. Then*

$$M_{min} = \cup_p P^*_{p,d}(I^*(p))$$

where $P_{p,d}$ is the set of fully local deductive rules in $\widetilde{\Gamma}^*(p) \cup \cup_q \Gamma^*(q, p)$.

Proof. Let $S_0 = (I_0, \Gamma_0, \widetilde{\Gamma}_0)$ be a positive initial state with a finite number of peers which converges to the (finite) state $S_\infty = (I_\infty, \Gamma_\infty, \widetilde{\Gamma}_\infty)$. Let M_{min} be the unique minimal model of the Datalog program $c(I_0, \Gamma_0)$. Given a run $(I_0, \Gamma_0, \widetilde{\Gamma}_0)$, $(I_1, \Gamma_1, \widetilde{\Gamma}_1)$, $(I_2, \Gamma_2, \widetilde{\Gamma}_2) \ldots$, we use $P_{p,d,i}$ (resp. $P_{p,a,i}$) to refer to the set of fully local deductive (resp. local active) rules in $\Gamma_i(p) \cup \cup_q \widetilde{\Gamma}_i(q, p)$. For ease of reference, we denote by F_i the set of facts $\cup_p P^*_{p,d,i}(I_i(p))$. Our aim is to show that $M_{min} = F_\infty$.

First direction: $F_\infty \subseteq M_{min}$
Consider the run $r = (I_0, \Gamma_0, \widetilde{\Gamma}_0), (I_1, \Gamma_1, \widetilde{\Gamma}_1), (I_2, \Gamma_2, \widetilde{\Gamma}_2) \ldots$. Let p_i be the peer whose move takes the state $(I_i, \Gamma_i, \widetilde{\Gamma}_i)$ to $(I_{i+1}, \Gamma_{i+1}, \widetilde{\Gamma}_{i+1})$. We will show by induction on i that (a) $P^*_{p_i,d,i}(I_i(p_i)) \subseteq M_{min}$, (b) $P_{p_i,a,i}(P^*_{p_i,d,i}(I_i(p_i))) \subseteq M_{min}$,

and (c) $M_{min} \models \widetilde{\Gamma}_{i+1}(p_i, q)$ for all $q \neq p_i$. Because of the monotonicity of states in r (cf. proof of Theorem 3.8), it follows from (a) and our definition of the sets F_i that $F_\infty \subseteq M_{min}$. Consider first the base case ($i = 0$). For (a), we note that $I_0(p_0) \cup \Gamma_0(p_0) \subseteq c(I_0, \Gamma_0)$ and $\cup_q \widetilde{\Gamma}_0(q, p_0) = \varnothing$ (since (I_0, Γ_0) is an initial state). We can thus deduce that $P^*_{p_0,d,0}(I_0(p_0)) \subseteq M_{min}$. For (b), we use (a) and the fact that $P_{p_0,a,0} \subseteq \Gamma_0(p_0)$ (as there are no delegations in the first time step). For (c), we first note that rules in $\widetilde{\Gamma}_1(p_0, q)$, are known to be of one of two types. The first type of rules are of the form

$$vA \text{ :- } vM@vQ(v\overline{U}), v\Theta_1$$

where A :- $\Theta_0, M@Q(\overline{U}), \Theta_1$ is a rule in $P_{p_0,a,0}$ and v is a valuation such that $v\Theta_0$ holds in $P^*_{p_0,d,0}(I_0(p_0))$ and $vQ = q(\neq p_0)$. In this case, the fact that $P^*_{p_0,d,0}(I_0(p_0)) \subseteq M_{min}$ ensures that $v\Theta_0$ holds in M_{min}. Since we also have $P_{p_0,a,0} \subseteq c(I_0, \Gamma_0)$, all rules in $P_{p_0,a,0}$ must holds in M_{min}, which means the partially instantiated rule vA :- $vM@vQ(v\overline{U}), v\Theta_1$ must also be satisfied by M_{min}. All other rules in $\widetilde{\Gamma}_1(q, p_0)$ are of the form vA :- where A :- Θ is a rule in $P_{p_0,a,0}$ and v is a valuation such that $v\Theta$ holds in $P^*_{p_0,d,0}(I_0(p_0))$ and $vA = r@q(\overline{u})$ for some $r \in \mathcal{I}$. Again, the fact that $P^*_{p_0,d,0}(I_0(p_0)) \subseteq M_{min}$ means that $v\Theta$ holds in M_{min}, and the fact that $P_{p_0,a,0} \subseteq c(I_0, \Gamma_0)$ means that vA :- must hold in the minimal model M_{min}.

For the induction step, suppose our claim holds for $i \leqslant k$. Let j be such that $p_j = p_{k+1}$ and $p_{j'} \neq p_{k+1}$ for all $j < j' < k + 1$, or 0 in the case where p_j has never been visited. Then it follows from our definition of moves and runs that

$$I_{k+1}(p_{k+1}) \subseteq I_j(p) \ \cup \bigcup_{j<l<k+1} P_{p_l,a,l}(P^*_{p_l,d,l}(I_l(p_l)))$$

It follows then from part (b) of the IH applied to timepoints $j, j + 1, \ldots, k$ that $I_{k+1}(p_{k+1}) \subseteq M_{min}$. Part (c) of the IH applied to the timepoints in which a peer $q \neq p_{k+1}$ was last visited gives us $M_{min} \models \cup_q \widetilde{\Gamma}_{k+1}(q, p_{k+1})$. Together with the fact that $\Gamma_{k+1}(p_{k+1}) = \Gamma_0(p_{k+1}) \subseteq c(I_0, \Gamma_0)$, we obtain

$$M_{min} \models P_{p_{k+1},a,k+1} \cup P_{p_{k+1},a,k+1}$$

Parts (a) and (b) of our claim follow directly. Now for part (c), consider some rule in $\widetilde{\Gamma}_{k+2}(p_{k+1}, q)$. First consider the case where the rule is of the form

$$vA \text{ :- } vM@vQ(v\overline{U}), v\Theta_1$$

where A :- $\Theta_0, M@Q(\overline{U}), \Theta_1$ is a rule in $P_{p_{k+1},a,k+1}$ and v is a valuation such that $v\Theta_0$ holds in $P^*_{p_{k+1},d,k+1}(I_{k+1}(p_{k+1}))$ and $vQ = q(\neq p_{k+1})$. We know $P^*_{p_{k+1},d,k+1}(I_{k+1}(p_{k+1})) \subseteq M_{min}$ from part (a), so $v\Theta_0$ must hold in M_{min}. This

together with the fact (from above) that $M_{min} \models P_{p_{k+1},a,k+1}$ means the partially instantiated rule vA :- $vM@vQ(v\overline{U}), v\Theta_1$ must also be satisfied by M_{min}. Suppose instead our rule is of the form vA :- where A :- Θ is a rule in $P_{p_{k+1},a,k+1}$ and v is a valuation such that $v\Theta$ holds in $P^*_{p_{k+1},d,k+1}(I_{k+1}(p_{k+1}))$ and $vA = r@q(\overline{u})$ for some $r \in \mathcal{I}$. We again utilize the fact that $P^*_{p_{k+1},d,k+1}(I_{k+1}(p_{k+1})) \subseteq M_{min}$ and $M_{min} \models P_{p_{k+1},a,k+1}$, which give $v\Theta \subseteq M_{min}$ and hence $M_{min} \models vA$:-.

Second direction: $M_{min} \subseteq F_\infty$
We proceed by induction on the depth of proof trees for facts in M_{min}. The base case is when the proof tree of a fact $r@p(\overline{u}) \in M_{min}$ has depth 0, i.e., it appears explicitly in $c(I_0, \Gamma_0)$. There are two possibilities: either $r@p(\overline{u}) \in I_0(p)$ or the rule $r@p(\overline{u})$:- appears in some $\Gamma_0(q)$. In the former case, monotonicity (cf. proof of Theorem 3.8) ensures that $r@p(\overline{u}) \in I_\infty(p) \subseteq F_\infty$. In the latter case, if $r@p$ is extensional, then $r@p(\overline{u})$ will be sent to p the first time q is visited and will remain at p by monotonicity. If $r@p$ is an intensional relation name and $q = p$, then $r@p(\overline{u})$:- belongs to $P_{p,d,\infty}$. If $q \neq p$, then $r@p(\overline{u})$:- will be delegated to p every time q is visited, and hence will belong to $\widetilde{\Gamma}_\infty(q, p)$, and hence to $P_{p,d,\infty}$. In all cases, we obtain $r@p(\overline{u}) \in \cup_p P^*_{p,d,\infty}(I_\infty(p)) = F_\infty$.

For the induction step, suppose that all facts in M_{min} with proof trees of depth at most k appear in F_∞. Consider some fact $r@p^*(\overline{u})$ with a proof tree of depth $k+1$. Then there must exist some rule

$$\alpha = M_{n+1}@Q_{n+1}(\overline{U}_{n+1}) \text{ :- } M_1@Q_1(\overline{U}_1)...M_n@Q_n(\overline{U}_n)$$

in $\cup_p \Gamma(p)$ and some valuation v such that

$$r@p^*(\overline{u}) = vM_{n+1}@vQ_{n+1}(v\overline{U}_{n+1})$$

and for all $1 \leqslant j \leqslant n$, the fact

$$s_j@q_j(\overline{t_j}) = vM_j@vQ_j(\overline{U}_j)$$

possesses a proof tree of depth at most k. Consider some run $r = (I_0, \Gamma_0, \widetilde{\Gamma}_0)$, $(I_1, \Gamma_1, \widetilde{\Gamma}_1), (I_2, \Gamma_2, \widetilde{\Gamma}_2) \ldots$ of (I_0, Γ_0). Applying the IH, we obtain $s_j@q_j(\overline{t_j}) \in F_\infty$ for all $1 \leqslant j \leqslant n$. It follows that we can find some index m such that $s_j@q_j(\overline{t_j}) \in F_m$ for all $1 \leqslant j \leqslant n$. Because all runs of (I_0, Γ_0) converge to the same state, we can assume without loss of generality that it is a q_j-move which takes the state $(I_{m+j-1}, \Gamma_{m+j-1}, \widetilde{\Gamma}_{m+j-1})$ in r to the state $(I_{m+j}, \Gamma_{m+j}, \widetilde{\Gamma}_{m+j})$, for all $1 \leqslant j \leqslant n$. We aim to show that $r@p^*(\overline{u}) \in F_{m+n}$, hence $r@p^*(\overline{u}) \in F_\infty$. We first remark that for all peers p, the set $P^*_{p,d,m}(I_m(p))$

can only consist of p-facts. This is because $I_0(p)$ contains only p-facts (by definition), only p-facts are added to $I_i(p)$ (by definition of moves), and $P_{p,d,m}$ consists of only deductive rules in p, i.e., rules using intensional p-relations. It follows then that $s_j@q_j(\overline{t_j}) \in P^*_{q_j,d,m}(I_m(q_j))$ for all $1 \leqslant j \leqslant n$. The safety condition implies that the term Q_1 equals a peer constant q_1. We can suppose that at timepoint m, $\alpha \in \Gamma_m(q_1)$.

Then it is q_1's move. If α is fully local deductive for q_1, then p^* and all of the q_j must be equal to q_1. This means that $s_j@q_j(\overline{t_j}) \in P^*_{q_1,d,m}(I_m(q_1))$ for all j, and so $r@p^*(\overline{u}) \in P^*_{q_1,d,m}(I_m(q_1))$. Thus, $r@p^*(\overline{u}) \in F_m$, and by monotonicity of states, $r@p^*(\overline{u}) \in F_{m+n}$. Next consider the more interesting case where α is not a fully local deductive rule for q_1. Let l be the maximal index such that $q_j = q_1$ for all $1 \leqslant j \leqslant l$. Then we have $s_j@q_j(\overline{t_j}) \in P^*_{q_1,d,m}(I_m(q_1))$ for all $1 \leqslant j \leqslant l$. If $l = n$, then $r@p^*(\overline{u}) \in I_{m+1}(p^*)$, and so again, by monotony, $r@p^*(\overline{u}) \in F_{m+n}$. If instead we have $l < n$, then delegation comes into play. Specifically, let v' be the minimal sub-valuation of v such that $v'M_j@v'Q_j(v'\overline{U_j}) = s_j@q_j(t_j)$ for all $1 \leqslant j \leqslant l$. Note that by the safety condition, Q_{l+1} must now be instantiated to q_l. It follows that the rule α'

$$v'M_{n+1}@v'Q_{n+1}(v'\overline{U}_{n+1}) \text{ :-}$$
$$v'M_l@v'Q_l(v'\overline{U}_l)...v'M_n@v'Q_n(v'\overline{U}_n)$$

must belong to $\widetilde{\Gamma}_{m+1}(q_1, q_l)$. By monotony, $\alpha' \in \widetilde{\Gamma}_{m+l-1}(q_1, q_l)$, and $s_j@q_j(\overline{t_j})$ $\in F_{m+l-1}$ We can thus repeat the same procedure to q_l when at timepoint $m + l - 1$ it is its turn to move. We will either finish (in which case the fact $r@p^*(\overline{u})$ is derived and preserved) or continue via delegations to the next peer, and so forth, until the final peer is treated and the fact $r@p^*(\overline{u})$ has been produced. We thus find the desired $r@p^*(\overline{u}) \in F_{m+n}$. □

3.4.2 Strongly-stratified *Webdamlog*

With negation, convergence is not guaranteed in the general case as illustrated by the following example.

Example 3.13. Consider the program that is stratified in the sense of Datalog with stratified negation:

```
intensional s@p, r@p, r@q
at p: r@q() :- r@p()
      r@p() :- ¬s@p()
at q: r@p() :- r@q()
      s@p() :-
```

Any run of this system that begins with p converges to a state where p delegates $r@q()$:- to q and q delegates $r@p()$:- and $s@p()$:- to p. On the other hand, runs that begin with q converge to a state where p delegates nothing to q and q delegates $s@p()$:- to p.

As already mentioned for the non-monotone updates in the previous subsection, one may adapt methods of view maintenance to solve the problem. We develop in this section an alternative in which syntactic restrictions prohibit circles of wrong deductions, without having to deal with the complexity of view maintenance in presence of belief revision. Note that most of the examples of the paper belong to (or are easily adapted to) this restricted class.

A *stratification* σ' is an assignment of numbers to relations, i.e., to pairs $r@p$. If $\sigma'(r@p) = i$, we say that $r@p$ is in the ith stratum. The stratification is *strong* if for each i, all the relations in the ith stratum refer to the same peer. Given a strong stratification σ', an instantiated rule is σ'-stratified if all relation names of positive body atoms appear in a stratum smaller or equal to that of the head relation and all relation names of negative terms belong to a strictly smaller stratum. Note that a stratification for Example 3.13 would not be strong because $r@p$ and $r@q$ have to be in the same stratum, although they belong to different peers.

In our setting, we see a strong stratification σ' of $\mathcal{I}$ as an extra component of the system's schema. The strong stratification works much like the typing constraint σ in that it tells us whether a particular rule instantiation is legal. Specifically, a peer is only allowed to use instantiated rules which are σ'-stratified. Observe that our use of stratification is in the spirit of classical Datalog with stratified negation, namely preventing cycling through negation. However, the way stratification is enforced is somewhat different. In the centralized context, one analyzes the program and checks for the existence of a stratification. In the distributed case, this is not possible because no one has access to the entire program. Also, the use of relation and peer variables makes such a computation even less conceivable. So, instead, one assumes that a stratification is imposed and the computation is such that it prevents deriving facts with rule instantiations that would violate the strong stratification.

There is a subtlety with strong stratification arising from general delegation. Indeed, we will see that the result does not hold for WL. So the next result deals simply with view delegation, i.e., the language VWL. One of the advantages of VWL is that at the time a rule is delegated, it is possible to check that it does not violate the strong stratification. We consider systems with finitely many peers, where the extensional facts are constant and only the intensional delegations vary. Formally, a *Webdamlog* system is said to be

strongly-stratified if for some strong stratification σ':

1. its local computation is constrained by the stratification σ'.

2. Each extensional relation $m@p$ is made persistent with a rule of the form $m@p(\overline{U})$:- $m@p(\overline{U})$ and these are the only active rules in the system[1]. We say the system is *purely intensional*.

Observe that, by Condition (2), the set of extensional facts is constant whereas it was increasing for positive systems. So Condition (2) here is more restrictive than for positive systems. Thus, strictly speaking the two classes are incomparable. Clearly, it would be interesting to consider classes that would include both.

We are now ready to present our results, following the same logic as in the previous section.

Theorem 3.14 (Convergence)**.** *All strongly-stratified VWL systems over a finite number of peers converge.*

Proof. Let us first remark that deductive rules in SWL can only be of two types: fully local deductive or local deductive. This means that the only types of rules that can be delegated to a peer p are fully instantiated body-less rules of the form $r@p(\overline{u})$:- . The general idea of the proof is as follows. Given a run, we will prove that for each stratum, there is a state after which the stratum has converged. A similar argument will prove that the limit is the same for each run.

Consider a σ'-stratified system (I_0, Γ_0) with rules in VWL and a finite number of peers. Let $r = (I_0, \Gamma_0, \varnothing)(I_1, \Gamma_1, \widetilde{\Gamma}_1)(I_2, \Gamma_2, \widetilde{\Gamma}_2)\ldots$ be a run of this system. For simplicity, in what follows, we use $P_{p,d,i}$ to refer to the set of fully local deductive rules in $\Gamma_i(p) \cup \cup_q \widetilde{\Gamma}_i(q,p)$.

First, we can show by induction that for all $i \geqslant 0$, every state $(I_i, \Gamma_i, \widetilde{\Gamma}_i)$ is intensional, $I_i = I_0$, and $\Gamma_i = \Gamma_0$. The base case $i = 0$ is immediate. For the induction step, suppose we have the result for $i < k$ and consider state $(I_k, \Gamma_k, \widetilde{\Gamma}_k)$ resulting from a p-move. From the IH, we know that $(I_{k-1}, \Gamma_{k-1}, \widetilde{\Gamma}_{k-1})$ is intensional, and so the only active rules in $\Gamma_{k-1}(p)$ and $\cup_q \widetilde{\Gamma}_{k-1}(q,p)$ are persistence rules for p's extensional predicates. We also have $\Gamma_k = \Gamma_0$ from the definition of runs. In particular, this means that $\Gamma_k(p)$ contains persistence rules for each of p's extensional predicates. This means that p copies its extensional facts ($I_k(p) = I_{k-1}(p)$) and does not send any extensional facts to other peers ($I_k(q) = I_{k-1}(q)$ for $q \neq p$). We thus have

[1]Technically speaking, if we want to use variable or peer relations in the rule heads, then we must forbid instantiations which yield extensional relations in the heads.

$I_k = I_{k-1} = I_0$. Finally, we note that $(I_{k-1}, \Gamma_{k-1}, \widetilde{\Gamma}_{k-1})$ contains no other active rules besides persistence rules, which means that all delegations will involve deductive rules.

Given the strong stratification σ', let us prove that for each stratum i, there is a timepoint $t_i \geqslant 0$ such that after each $t \geqslant t'$, the restriction of $\widetilde{\Gamma}_t$ to rules with head in strata less or equal to i is the same as the one of $\widetilde{\Gamma}_{t_i}$. Let us start with the first stratum, call it 0. Suppose that p^* is the peer associated with this stratum. Let t_0 be the first occurrence of a p^*-move after visiting all the other peers. Such a timepoint must exist since the number of peers is finite (this is assumed in the statement of the theorem) and the run is fair. We claim that t_0 has the desired properties. Consider some timepoint $t \geqslant t_0$ in which it's peer q's turn to move and some delegation appearing in $\widetilde{\Gamma}_{t+1}(q, p^*)$. We remark that because we only have VWL rules, the delegation must be of the form $r@p^*(\overline{u})$:-. To produce this delegation, there must be a rule in $\Gamma_t(q) = \Gamma_0(q)$ of the following form

$$M_{n+1}@Q(\overline{U}_{n+1}) \text{ :-}$$
$$(\neg)M_1@q(\overline{U}_1), (\neg)M_2@q(\overline{U}_2), ...(\neg)M_n@q(\overline{U}_n)$$

and some valuation v satisfying the typing σ and stratification σ' such that: $vM_{n+1}@vQ(v\overline{U}_{n+1}) = r@p^*(\overline{u})$, each positive body fact $vM_i@q(v\overline{U}_i)$ belongs to $P^*_{q,d,t}(I_t(q))$, and each negated body fact $\neg vM_i@q(v\overline{U}_i)$ is such that $vM_i@q(v\overline{U}_i)$ is not in $P^*_{q,d,t}(I_t(q))$. We note however that because v satisfies the strong stratification, we are at peer $q \neq p$, and the head relation $r@p$ is in the lowest stratum, all relations $vM_i@q$ must be extensional. As the extensional facts of each peer are the same at each timepoint (see above), it follows that this delegation is produced at each and every visit to q, and in particular the very first visit to q, which occurs before t_0. Thus, this delegation already appears in $\widetilde{\Gamma}_{t_0}(q, p^*)$. A very similar argument shows that every delegation concerning stratum 0 which appears in $\widetilde{\Gamma}_{t_0}(q, p^*)$ also appears in $\widetilde{\Gamma}_t(q, p^*)$ for all $t \geqslant t_0$.

Now let us consider higher strata. Suppose our claim holds for strata up to and including k. This means we can find a timepoint t_k such that for all $t \geqslant t_k$, the restriction of $\widetilde{\Gamma}_t$ to rules with head in strata less or equal to k is the same as the one of $\widetilde{\Gamma}_{t_k}$. Again, we use p^* to refer to the peer associated with the stratum of interest (here $k+1$). Set t_{k+1} equal to the timepoint after t_k in which we first visit p^* after having visited all other peers at least once since timepoint t_k. Consider some timepoint $t \geqslant t_0$ in which q moves and produces some delegation in $\widetilde{\Gamma}_{t+1}(q, p^*)$. Again, because we only have VWL rules, we know this delegation must be of the form $r@p^*(\overline{u})$:-. To produce it,

there must be a rule in $\Gamma_t(q) = \Gamma_0(q)$ of the following form

$$M_{n+1}@Q(\overline{U}_{n+1}) \text{ :-}$$
$$(\neg)M_1@q(\overline{U}_1), (\neg)M_2@q(\overline{U}_2), ...(\neg)M_n@q(\overline{U}_n)$$

and some valuation v satisfying the typing σ and stratification σ' such that: $vM_{n+1}@vQ(v\overline{U}_{n+1}) = r@p^*(\overline{u})$, each positive body fact $vM_i@q(v\overline{U}_i)$ belongs to $P^*_{q,d,t}(I_t(q))$, and each negated body fact $\neg vM_i@q(v\overline{U}_i)$ is such that $vM_i@q(v\overline{U}_i)$ is not in $P^*_{q,d,t}(I_t(q))$. Because v satisfies the strong stratification, we are at peer $q \neq p$, and the head relation $r@p$ is in the lowest stratum, we know all body facts $vM_i@q(v\overline{U}_i)$ must either be extensional or intensional but in a lower stratum ($\leqslant k$). We have already seen that extensional facts are fixed throughout the run. Since $t \geqslant t_{k+1} > t_k$, we know that all delegations for strata less than or equal to k are fixed and equal to those found at timepoint t_k. It follows that this delegation is produced at each and every visit to q following timepoint t_k, and hence in the visit to q between timepoints t_k and t_{k+1}. Thus, this delegation already appears in $\widetilde{\Gamma}_{t_{k+1}}(q, p^*)$. We can similarly show that all delegations stratum $k+1$ delegations in $\widetilde{\Gamma}_{t_{k+1}}(q, p^*)$ are also found in $\widetilde{\Gamma}_t(q, p^*)$ for all $t \geqslant t_{k+1}$.

We now prove that all systems converge to the same limit. In fact, we can straightforwardly extend the previous proof by adding to the claim that each stratum $k+1$ converges to the same value on all runs. In the base case, we use the fact that the extensional facts are the same in all runs. This means delegations for the first stratum will be the same for all runs. For later strata, we use the fact that the delegations at level $k+1$ are fully determined by the delegations in previous strata. □

This result does not hold if we allow general delegation instead of view delegation. This is because with general delegation, a peer p may delegate a partially instantiated rule to q. As the relation and peer terms of the rule may contain variables, peer p may not be able to decide whether the rule is σ'-stratified, and neither will q (or later peers) as they do not know which relations p used to launch the delegation. So enforcement of the stratification is not straightforward. This is illustrated by the following example.

Example 3.15. Consider the following program:

```
intensional m@p, s@q, r@q
at p:  m@p($x) :- m@p($x), r@q($x)
       m@p($x) :- r@q($x), ¬s@q()
at p': s@q() :-
at q:  r@q(a) :-
```

Consider a run that starts by firing p, q, then p. Then the rule $m@p(a)$:- is delegated by q to p and will remain forever. Now, consider a run that starts by firing p'. Then q will know $s@q()$:-. from the beginning and will never delegate $m@p(a)$:-.

Convergence also holds for strongly-stratified VWL systems in the presence of insertions as well as deletions.

Theorem 3.16 (Update)**.** *Let (I,Γ) be a VWL system with strong stratification σ' over a finite number of peers. Consider $(I^+,I^-,\Gamma^+,\Gamma^-)$ where I^+, I^- are sets of extensional facts and Γ^+,Γ^- are sets of deductive rules. For each run of the system (I,Γ), if for some k a given state $(I_k,\Gamma_k,\widetilde{\Gamma}_k)$ is replaced by $(I_k \cup I^+ \setminus I^-,\ \Gamma_k \cup \Gamma^+ \setminus \Gamma^-,\widetilde{\Gamma}_k)$, then the modified run converges to the convergence state of the σ'-stratified system $(I \cup I^+ \setminus I^-,\ \Gamma \cup \Gamma^+ \setminus \Gamma^-)$.*

Proof. First, it is straightforward to show that $(I \cup I^+ \setminus I^-,\Gamma \cup \Gamma^+ \setminus \Gamma^-)$ respects the constraints of intensional states. Let us recall from the proof of Theorem 3.14 that until the insertion point k, $I_k = I$ and $\Gamma_k = \Gamma$. So at the end of the timepoint k, the state is indeed $(I \cup I^+\ I^-, \Gamma \cup \Gamma^+ \setminus \Gamma^-, \widetilde{\Gamma}_k)$. Then observe that the proof never used the fact that $\widetilde{\Gamma}$ was initially empty, except to prove that the initial state was intensional. So the proof applies as it is and gives the desired result. □

This theorem can obviously be generalized to any sequence of updates. The final theorem of this section shows that the set of facts computed by a σ'-stratified system corresponds to the set of facts in the minimal model of a centralized version of the system. As in the previous section, we associate a σ'-stratified *Webdamlog* system (I,Γ) with the set $\cup_p(I(p) \cup \Gamma(p))$ composed of the facts and rules of all peers. We then transform this set of facts and rules into a standard Datalog program by instantiating the variable predicates in the rules and removing rules which violate the typing constraints σ or the strong stratification σ'. We use $c_s(I,\Gamma)$ to refer to the resulting Datalog program.

Theorem 3.17 (Centralized)**.** *Let (I,Γ) be a σ'-stratified system with a finite number of peers and rules in SWL, which converges to $(I^*,\Gamma^*,\widetilde{\Gamma}^*)$, and let M_{min} be the unique minimal model of the Datalog program $c_s(I,\Gamma)$. Then*

$$M_{min} = \cup_p P^*_{p,d}(I^*(p))$$

where $P^*_{p,d}$ is the set of fully local deductive rules in $\widetilde{\Gamma}^*(p) \cup \cup_q \Gamma^*(q,p)$.

Proof. Let $S_0 = (I_0,\Gamma_0,\widetilde{\Gamma}_0)$ be a strongly stratified VWL system (with strong stratification σ') which converges to the finite state $S_\infty = (I_\infty,\Gamma_\infty,\widetilde{\Gamma}_\infty)$. As

the rules in the Datalog program $c_s(I_0, \Gamma_0)$ are stratified with respect to σ' (by construction), we can be sure that there is a unique minimal model of $c_s(I_0, \Gamma_0)$. We use M_{min} to denote this minimal model. Given a run $(I_0, \Gamma_0, \widetilde{\Gamma}_0), (I_1, \Gamma_1, \widetilde{\Gamma}_1), (I_2, \Gamma_2, \widetilde{\Gamma}_2) \ldots$ of our system, we use $P_{p,d,i}$ to refer to the set of fully local deductive rules in $\Gamma_i(p) \cup \cup_q \widetilde{\Gamma}_i(q, p)$. For ease of reference, we denote by F_i the set of facts $\cup_p P^*_{p,d,i}(I_i(p))$. Our aim is to show that $M_{min} = F_\infty$.

We first note that the desired equality holds if we consider only extensional facts. This is because the only rules with extensional heads in Γ_0 are extensional persistence rules. Thus, the extensional facts in F_∞ are precisely the original extensional facts $\cup_p I_0(p)$. The Datalog program $c_s(I_0, \Gamma_0)$ will contain these extensional facts, and will not contain any rules to create new extensional facts, so the extensional facts in M_{min} will be exactly $\cup_p I_0(p)$.

It thus remains to show the equality for intensional facts. The proof will proceed by induction on the strata of facts. In what follows, we will use the integers $0, 1, 2, \ldots$ to label the strata, with 0 being the lowest stratum. Also, given a set S of facts, we denote by $S[i]$ the set of facts whose relations belong to strata lower than or equal to i.

Base Case: $M_{min}[0] = F_\infty[0]$

First direction ($F_\infty[0] \subseteq M_{min}[0]$). Let us consider some intensional fact $r@p(\overline{u})$ from stratum 0 which belongs to F_∞, and hence more precisely to $P^*_{p,d,\infty}(I_\infty(p))$. We know that the set $P_{p,d,\infty}$ consists of fully local deductive rules from $\Gamma_\infty(p) = \Gamma_0(p)$ and delegated body-less rules $\cup_q \widetilde{\Gamma}_\infty(q, p)$. Moreover, we have seen in the proof of Theorem 3.14 that each body-less delegation with head relation in stratum 0 from a peer q results from evaluating the extensional q-facts present in the initial state using the instantiation of a local rule in Γ_q which respects σ and σ'. As the extensional q-facts in M_{min} are precisely those found in the initial state, and all well-typed rules from $\Gamma_0(q)$ respecting σ' can be found in $c_s(I_0, \Gamma_0)$, it follows that the delegated rule is entailed by M_{min}. Thus, all (well-typed and properly stratified) instantiations of rules in $P_{p,d,\infty}$ with heads of stratum 0 are entailed by M_{min}, and so are all extensional facts in $I_\infty(p)$. It follows that the fact $r@p(\overline{u})$ must belong to M_{min}.

Second direction ($M_{min}[0] \subseteq F_\infty[0]$). Consider some intensional fact $r@p(\overline{u})$ from stratum 0 which belongs to M_{min}. The proof proceeds by induction on the depth of the proof tree of $r@p(\overline{u})$. The base case is when $r@p(\overline{u})$ has depth 0, i.e., it appears explicitly in $c_s(I_0, \Gamma_0)$. There are two possibilities:

either $r@p(\overline{u}) \in I_0(p)$ or the rule $r@p(\overline{u})$:- appears in some $\Gamma_0(q)$. In the former case, we know from the proof of Theorem 3.14 that $I_\infty = I_0$, so we must have $r@p(\overline{u}) \in F_\infty$. In the latter case, as we are in an intensional system, the rule $r@p(\overline{u})$:- must be deductive. Either this rule appears in $\Gamma_0(p)$ (hence $\Gamma_\infty(p)$) or it will be delegated to p by another peer q at every visit to q, and thus will appear in $\widetilde{\Gamma}_\infty(q,p)$. In both cases, the rule must belong to $P_{p,d,i}$, hence $r@p(\overline{u}) \in P^*_{p,d,i}(I_i(p)) \subseteq F_\infty$. Now suppose the proof tree of fact $r@p(\overline{u})$ has depth $d+1$, and we already have the result for facts of stratum 0 with proof trees of depth at most d. Let β be the rule in $c_s(I_0, \Gamma_0)$ which was used for the last step of the proof of $r@p(\overline{u})$. As (I_0, Γ_0) is an intensional VWL system, it follows that all rules in (I_0, Γ_0) are of one of two types: persistence rules for extensional predicates, or local deductive rules. Thus, the rule β must be of the form

$$vM_{n+1}@vQ(v\overline{U}_{n+1}) \text{ :-}$$
$$(\neg)vM_1@q(v\overline{U}_1), (\neg)vM_2@q(v\overline{U}_2), ...(\neg)vM_n@q(v\overline{U}_n)$$

for some rule ρ

$$M_{n+1}@Q(\overline{U}_{n+1}) \text{ :-}$$
$$(\neg)M_1@q(\overline{U}_1), (\neg)M_2@q(\overline{U}_2), ...(\neg)M_n@q(\overline{U}_n)$$

in $\Gamma_0(q)$ and some valuation v which respects the typing constraints σ and the strong stratification σ', and is such that $vM_{n+1}@vQ = r@p$. Note in particular that this means that each of the (ground) relations $vM_j@q$ must be extensional or belong to the same stratum (0) as $r@p$. If there are any facts from the stratum 0 in the body, then they must use a relation with peer p, and so we would have $q = p$ (since only local deductive rules are permitted). Otherwise, if $q \neq p$, then only extensional relations may be used in the body. Also note that all atoms in the body which belong to stratum 0 must not be negated. We know that the rule β was used to derive the fact $r@p(\overline{u})$. This means that there must be a second valuation v' such that $v'vM_{n+1}@v'vQ(v'v\overline{U}_{n+1}) = r@p(\overline{u})$ and each literal $(\neg)vM_i@q(v'v\overline{U}_i)$ is either extensional and satisfied by the set of extensional facts or a positive atom of stratum 0 which has a proof tree of depth at most k. As F_∞ and M_{min} agree on all extensional facts, all extensional literals $(\neg)vM_i@q(v'v\overline{U}_i)$ are satisfied by $P^*_{q,d,\infty}(I_\infty(q))$. For the remaining body atoms, we use the IH to infer that each atom $vM_i@q(v'v\overline{U}_i)$ of stratum 0 belongs to F_∞, and more specifically to $P^*_{q,d,\infty}(I_\infty(q))$. If $q = p$, then we can use the rule ρ in $\Gamma_\infty(p) = \Gamma_0(p)$ together with the valuation $v'' = v'v$ and the facts $vM_i@p(v'v\overline{U}_i) \in P^*_{p,d,\infty}(I_\infty(p))$ to obtain $r@p(\overline{u}) \in P^*_{p,d,\infty}(I_\infty(p))$. If $q \neq p$, then we know from above that each

$vM_i@q(v'v\overline{U}_i)$ must be an extensional fact and it must belong to $P^*_{q,d,\infty}(I_\infty(q))$. It follows that q must delegate the rule $r@p(\overline{u})$:- to p. The fact that the run has converged to $(I_\infty, \Gamma_\infty, \widetilde{\Gamma}_\infty)$ means that this delegation must appear in $\widetilde{\Gamma}_\infty)$. It follows that $r@p(\overline{u})$:- belongs to $P_{p,d,i}$, hence $r@p(\overline{u}) \in P^*_{p,d,i}(I_i(p)) \subseteq F_\infty$.

Induction Step: show $M_{min}[k+1] = F_\infty[k+1]$ assuming $M_{min}[k] = F_\infty[k]$

First direction ($F_\infty[k+1] \subseteq M_{min}[k+1]$). We suppose that $F_\infty[k] \subseteq M_{min}[k]$. Let us consider some intensional fact $r@p(\overline{u})$ from stratum $k+1$ which belongs to F_∞, and hence to $P^*_{p,d,\infty}(I_\infty(p))$. We know that the set $P_{p,d,\infty}$ consists of fully local deductive rules from $\Gamma_\infty(p) = \Gamma_0(p)$ and delegated body-less rules from $\cup_q \widetilde{\Gamma}_\infty(q,p)$. As for the delegated rules, note that if $s@p(\overline{w})$:- appears in $\widetilde{\Gamma}_\infty(q,p)$, there must exist a rule in $\Gamma_\infty(q) = \Gamma_0(q)$ of the form

$$M_{n+1}@Q(\overline{U}_{n+1}) :-$$
$$(\neg)M_1@q(\overline{U}_1), (\neg)M_2@q(\overline{U}_2), ...(\neg)M_n@q(\overline{U}_n)$$

and a valuation v satisfying the typing σ and strong stratification σ' such that: $vM_{n+1}@vQ(v\overline{U}_{n+1}) = s@p(\overline{w})$, each fact $vM_i@q(v\overline{U}_i)$ appearing positively in the body belongs to $P^*_{q,d,\infty}(I_\infty(q))$ (and hence to F_∞), and each negated fact $\neg vM_i@q(v\overline{U}_i)$ in the body does not appear in $P^*_{q,d,\infty}(I_\infty(q))$ (nor a fortiori in F_∞). Because v respects the strong stratification σ', and $q \neq p$, we know that every relation $vM_i@q$ is either extensional or must belong to a stratum k or less. From the IH, we know that M_{min} and F_∞ agree on all intensional facts appearing in strata up to and including k, and we have seen earlier in the proof that the same is true for extensional facts. It follows that each fact $vM_i@q(v\overline{U}_i)$ appearing positively in the body belongs to M_{min}, and each negated fact $\neg vM_i@q(v\overline{U}_i)$ in the body does not appear in M_{min}. Moreover, we know that the instantiated rule used to produce the delegation is entailed by M_{min}. Thus, we have that M_{min} entails the delegation $s@p(\overline{w})$:-. Thus, all (well-typed and properly stratified) instantiations of rules in $P_{p,d,\infty}$ whose head relations are in strata at $k+1$ are entailed by M_{min}. Moreover, we know that only (well-typed and stratified) instantiations of rules in $P_{p,d,\infty}$ with head relations in stratum $k+1$ or lower are used in the production of $r@p(\overline{u})$. Finally, we know that all extensional p-facts in $I_\infty(p) = I_0(p)$ belong to M_{min}. It follows that the fact $r@p(\overline{u})$ belongs to M_{min}.

Second direction ($M_{min}[k+1] \subseteq F_\infty[k+1]$). Consider some intensional fact $r@p(\overline{u}) \in M_{min}$ from the stratum $k+1$. As σ' provides a stratification of $c_s(I_0, \Gamma_0)$, it is possible to find a proof tree for $r@p(\overline{u})$ whose leaves use only (negations of) facts in M_{min} belonging to strata $\leqslant k$. We will thus again

proceed by induction on the depth of such a proof tree. The base case is when the proof tree for $r@p(\overline{u})$ has depth 0, i.e., it appears explicitly in $c_s(I_0, \Gamma_0)$. We can then proceed as in the base case for stratum 0. Suppose next that we have already shown the result for intensional facts in M_{min} belonging to stratum $k+1$ and having proof trees from facts in strata $\leqslant k$ of depth at most d. Consider $r@p(\overline{u}) \in M_{min}$ from the stratum $k+1$ with a proof tree of depth $d+1$. Let β be the rule in $c_s(I_0, \Gamma_0)$ which was used for the last step of the proof. As we saw earlier, β must be of the form

$$vM_{n+1}@vQ(v\overline{U}_{n+1}) \text{ :-}$$
$$(\neg)vM_1@q(v\overline{U}_1), (\neg)vM_2@q(v\overline{U}_2), ...(\neg)vM_n@q(v\overline{U}_n)$$

for some rule ρ

$$M_{n+1}@Q(\overline{U}_{n+1}) \text{ :-}$$
$$(\neg)M_1@q(\overline{U}_1), (\neg)M_2@q(\overline{U}_2), ...(\neg)M_n@q(\overline{U}_n)$$

in $\Gamma_0(q)$ and some valuation v which respects the typing constraints σ and the strong stratification σ' and such that $vM_{n+1}@vQ = r@p$. It follows that each (ground) relation $vM_i@q$ is either extensional or an intensional relation which belongs to a stratum lower than or equal to $k+1$. We also know that β was used to derive the fact $r@p(\overline{u})$, which implies the existence of a second valuation v' such that $v'vM_{n+1}@v'vQ(v'v\overline{U}_{n+1}) = r@p(\overline{u})$ and each literal $(\neg)vM_i@q(v'v\overline{U}_i)$ is either (i) a (possibly negated) extensional fact which is satisfied by M_{min}, (ii) a (possibly negated) intensional fact from some stratum $\leqslant k$ which holds in M_{min}, or (iii) a non-negated intensional fact from stratum $k+1$ with a proof tree of depth at most k. We know from earlier in the proof that F_∞ and M_{min} agree on extensional facts. This means that every non-negated extensional fact $vM_i@q(v'v\overline{U}_i)$ belongs to F_∞ (more precisely $P^*_{q,d,\infty}(I_\infty(q))$) and every negated extensional fact $\neg vM_i@q(v'v\overline{U}_i)$ does not belong to $P^*_{q,d,\infty}(I_\infty(q))$.

For intensional facts from lower strata (k or less), we use the induction hypothesis (from the initial induction over strata) to obtain $F_\infty[k] = M_{min}[k]$. From this we can deduce that an intensional fact $vM_i@q(v'v\overline{U}_i)$ of stratum $\leqslant k$ which appears positively in the body of our rule must belong to F_∞ (or more specifically $P^*_{q,d,\infty}(I_\infty(q))$), and if it appears negatively in the rule, then it will not belong to $P^*_{q,d,\infty}(I_\infty(q))$.

Finally, if we have a non-negated intensional fact $vM_i@q(v'v\overline{U}_i)$ from stratum $k+1$ with a proof tree of depth at most k, then using the (local) IH, we obtain $vM_i@q(v'v\overline{U}_i) \in F_\infty$, and hence $vM_i@q(v'v\overline{U}_i) \in P^*_{q,d,\infty}(I_\infty(q))$. If we are in the case where $p = q$, then we can use the rule ρ in $\Gamma_\infty(p) =$

$\Gamma_0(p)$ together with the valuation $v'' = v'v$ and the facts $vM_i@p(v'v\overline{U}_i) \in P^*_{p,d,\infty}(I_\infty(p))$ to obtain $r@p(\overline{u}) \in P^*_{p,d,\infty}(I_\infty(p) \subseteq F_\infty)$. If $q \neq p$, then because we respect the strong stratification, we know that each $vM_i@q(v'v\overline{U}_i)$ must be either an extensional fact or an intensional fact from a stratum $\leqslant k$. In both cases, we have shown above that $vM_i@q(v'v\overline{U}_i)$ belongs to $P^*_{q,d,\infty}(I_\infty(q))$ when $vM_i@q(v'v\overline{U}_i)$ appears positively in the rule, and $vM_i@q(v'v\overline{U}_i)$ does not belong to $P^*_{q,d,\infty}(I_\infty(q))$ when it is appears negatively. Thus, the body of the rule is satisfied by $P^*_{q,d,\infty}(I_\infty(q))$. It follows that q must delegate the rule $r@p(\overline{u})$:- to p. The fact that the run has converged to $(I_\infty, \Gamma_\infty, \widetilde{\Gamma}_\infty)$ means that this delegation must appear in $\widetilde{\Gamma}_\infty$). It follows that $r@p(\overline{u})$:- belongs to $P_{p,d,i}$, hence $r@p(\overline{u}) \in P^*_{p,d,i}(I_i(p)) \subseteq F_\infty$. $\square$

Chapter 4

Webdamlog rule engine

In the present chapter, we consider the management of data and knowledge (i.e., programs) over a network of autonomous peers using the deduction supported by a *Webdamlog* rule engine. From a system viewpoint, the different actors are autonomous and heterogeneous in the style of P2P [AP07a, FHM05]. However, we do not see the system we developed as an alternative for managing information to existing centralized network services such as Facebook or Flickr. Rather, we view the system as the means of seamlessly integrating distributed knowledge residing in any of these services, as well as in a wide variety of systems managing personal or social data. The system takes advantage of a datalog engine to implement the *Webdamlog* language of Chapter 3, to support the distribution of both *data* and *knowledge* (i.e., programs) over a network of autonomous peers. The main contribution is our implementation of an engine to process efficiently *Webdamlog*, introduced in [7] and shown in a demonstration in [4]

Organization The chapter is organized as follows. In Section 4.1, we motivate our choice for the datalog engine *Bud*. In Section 4.2, we explain the implementation on top of *Bud* and slight departures from the model previously introduced in Section 3.1. Then in Section 4.3, we show how to apply known optimization techniques to *Webdamlog*. Also we introduce, in Section 4.4, a novel optimization technique for highly-dynamic programs. In the last section 4.5, we conclude with performance evaluation of the engine.

4.1 Datalog inside

Datalog evaluation has been intensively studied, and several open-source implementations are available. We chose not to implement yet another

datalog engine, but instead to extend an existing one. From the long list of engines still supported, see [LFWK09] for benchmarking of some of them, we hesitated between two open-source systems that are currently being supported, namely, *Bud* [ACHM11] from Berkeley University and *IRIS* [oI] from Innsbruck University.

- The *IRIS* system is implemented in Java and supports the main strategies for efficient evaluation of standard local datalog such as semi-naive evaluation [AHV95], Magic Sets [BMSU86] and Query Sub Query [Vie86]. Also it support negation.

- The *Bud* system also implements the semi-naive evaluation however it is implemented in the Ruby scripting language, which seemed less promising from a performance viewpoint. Nevertheless *Bud* provides technology for asynchronous communication between peers, hence it supports distributed datalog evaluation. And above all it focuses on non-monotonicity and provides efficient cache optimizations to support updates. That is an essential feature for *Webdamlog* extensional relations.

We finally decided in favor of *Bud*, both because of its support for asynchronous communication, and because its scalability has been demonstrated in real-life scenarios such as reimplementing with comparable performance Internet router and Hadoop File System as shown in [ACC+10, oUB]. In addition *Bud* is a very active project at this time and a follow-up of multiple previous successful prototypes such as P2 [LCH+05] from the Berkeley team. IRIS seemed less active since 2011 although it supports negation that *Bud* does not provide. We chose in favor of efficient distribution even if it meant giving up negation.

4.2 Connection between *Bud* and *Webdamlog*

4.2.1 *Webdamlog* computation on *Bud*

The *Bud* system supports a powerful datalog-like language introduced in [ACHM11]. Indeed, *Bud* is a distributed datalog engine with updates and asynchronous communications.

In the *Webdamlog* engine, a computation consists semantically of a sequence of *stages*, with each stage involving a single peer. Each stage of a *Webdamlog* peer computation is in turn performed by a three-step *Bud*

computation, described next. Note that we use the word *stage* for *Webdamlog* and *step* for *Bud*:

...	Stage at peer p			Stage at peer q			...
	Step 1	Step 2	Step 3	Step 1	Step 2	Step 3	

The 3 steps of a *Webdamlog* stage are as follows:

1. Inputs are collected including input messages from other peers, clock interrupts and host language calls.
2. Time is frozen; the union of the local store and of the batch of events received since the last stage is computed, and a *Bud* program is run to fixpoint.
3. Outputs are obtained as side effects of the program, including output messages to other peers, updates to the local store, and host language callbacks.

Observe that a fixpoint computation is performed at Step 2 by the local datalog engine (namely the *Bud* engine). This computation is based on a fixed program with no deletion over a fixed set of extensional relations and rules. In Step 3, deletion messages may be produced, along with updates to the set of rules and to the set of extensional relations (for different reasons, which we will explain further). Note that all this occurs *outside* the datalog fixpoint computation.

Relations appearing in the rules are implemented as *Bud* collections. Collections are the data structure for relations in *Bud* as in-memory key-values pairs. *Bud* distinguishes between three kinds of key-value sets:

1. A *table* collection stores a set of facts. A fact is deleted only when an explicit delete order is received. Tables are used to support *Webdamlog* extensional persistent relations. Remember that in the *Webdamlog* language in Section 3.1, a persistent relation is obtained by adding this rule:
$$r@p(\overline{\mathrm{U}}) \text{ :- } r@p(\overline{\mathrm{U}}), \neg del.r@p(\overline{\mathrm{U}}) \tag{4.1}$$
Hence, in the language, persistent or non-persistent relations differ only by the presence or absence of this rule. However in the implementation, these are completely different data structures. Thus a relation declared as persistent cannot be mutated into a non-persistent.

   In the implementation, due to the absence of support of negation in the *Bud* engine, Rule 4.1 is not evaluated during a single step of *Bud* computation but spans to two *Webdamlog* stages. This is a departure from the pure *Webdamlog* syntax that is caused by the use of *Bud*.

2. A *scratch* collection is used for storing results of intermediate computations. We use *scratch* collections to implement *Webdamlog* local intensional relations. The collections are emptied at Step 1 and receive facts during fixpoint computation at Step 2.

3. A *channel* collection provides support for asynchronous communications. It records facts that have to be sent to other peers. At Step 1, it contains all the messages and rules received from other peers since the last stage then it is emptied at Step 3. The channel mimics the behavior of non-persistent extensional relations of *Webdamlog* since it consumes the facts.

As in *Webdamlog*, facts in a peer are consumed by the engine at each firing of the peer (each stage). To make facts persistent, they have to be re-derived by the peer at each stage. This is captured in our implementation by assuming that rules re-derive extensional facts implicitly, unless a deletion message has been received.

We observe two subtleties that lead us to not fully adopt the original semantics of *Webdamlog*:

1. Since communications are asynchronous, there is no guarantee in *Webdamlog* as to when a fact written to a channel will be received by a remote peer. This is a departure from the original semantics of *Webdamlog*, which considered, for simplicity, that messages are transmitted instantaneously. We depart from the original semantics because it imposes some form of synchronization, that would drastically hinder performance.

2. A subtlety is that rules with variables as relation or peer names are not installed in one stage, they are processed in several stages to bound variables one by one. For non-local rules, delegations are created as stated in the model and sent to remote peers, however for local rules a delegation is sent to itself at a future stage. This is a slight departure from the original semantics of *Webdamlog* that we do not see as important.

4.2.2 Implementing Webdamlog rules

We now describe how *Webdamlog* rules are implemented on top of *Bud*. We Distinguish between 4 cases. This brings us to revisit the semantics of *Webdamlog* (from Chapter 3) with a focus on implementation. As in Chapter 3, whether a rule in a peer p is *local* (i.e., all relations occurring in the rule

body are p-relations) plays an important role. We consider 4 different cases of implementation for local-rules, depending on the type of the relation in the head, namely (A-D). We consider one case for non-local rules, namely (E). The last case (F) focuses on the use of variables for relation and peer names. For the first 5 cases, we ignore such variables.

A-B-C. Simple local rules In these three cases, the relations in the body are local, and depending on the type of the relations in the head, *Webdamlog* rules can be directly supported by simple translation into *Bud* rules:

A fully local deductive with local intensional head. It is standard local datalog evaluation.

B local updates with local extensional head. It is local active rules corresponding to datalog with updates. The non-monotonic extension of datalog.

C remote updates with remote extensional head. This corresponds to sending messages i.e. it is distributed datalog.

Note that, according to the semantic of *Webdamlog*, the behavior of these three different kinds of rules are not the same hence we use a different translation for each case. Let us consider a generic rule with relation h in the head and i p-relations $\mathsf{b_i}$ in the body. *Bud* provide three different operator to support these rules ; namely "<=", "<+" and "< ":

Instantaneous $\mathsf{h@p(X,Y)} <= \mathsf{b_1@p(X,Y)}, \ldots$ During Step 2, the rule are repeatedly evaluated i.e. facts derived during the current stage are reused until a fixpoint is reached. This corresponds to the **Case A** and intensional relations are materialized to fixpoint.

Deferred $\mathsf{h@p(X,Y)} <+ \mathsf{b_1@p(X,Y)}, \ldots$ The facts produced during Step 2 are inserted in the head collection for the next *Webdamlog* stage. Hence this is the immediate consequence operator. The main difference with the previous operator occurs especially when the rule is recursive. This implements **Case B**. Remark that if the head relation is a scratch collection, this operator will derive facts for the next stage that will be deleted at Step 1 of next stage according to the behavior of scratch collections. This operator has a counter part denoted with "<-" that sends delete messages for the next stage. Both operators are meant to deal with non-monotonicity that is why there effects occur outside the fixpoint computation. Notice that in *Bud* deletion messages (sent by "<-") are processed before insertion messages (sent by "<+").

Asynchronous $\mathsf{h@q(X,Y)} <\sim \mathsf{b_1@p(X,Y)}, \ldots$ In *Bud*, this operator is used to send facts to an external process: a terminal, a key-value store or communication channel. In the *Webdamlog* engine, the head relation will be a *Bud* channel collection connected to a remote peer. The facts produced during Step 2 will be sent via networking protocols, namely UDP in this implementation. A set of facts produced during Step 2 is written on the communication channel at Step 3. Due to asynchronism of network communication, *Bud* does not guarantee that two facts written by p at a given stage, will be received together. However we will see that *Webdamlog* engine will implement a mechanism to sends indivisible packets of facts at each stage. This implements the **Case C**.

D. Local with non-local intensional head Although it uses distributed datalog rules, *Bud* does not really support intensional relations. That is why from an implementation viewpoint, this case is the more complex. We illustrate it with an example. Consider an intensional relation $\mathsf{s_0@q}$ defined in the distributed setting by the following two rules:

[at p1] $\mathsf{s_0@q(X,Y)}$:- $\mathsf{r_1@p_1(X,Y)}$
[at p2] $\mathsf{s_0@q(X,Y)}$:- $\mathsf{r_1@p_2(X,Y)}$

Intuitively, the two rules specify a view relation $\mathsf{s_0@q}$ at q that is the union of two relations $\mathsf{r_1@p_1}$ and $\mathsf{r_1@p_2}$ from peers $\mathsf{p_1}$ and $\mathsf{p_2}$, respectively. Consider a possible naive implementation that would consist in materializing relation s_0 at q, and having $\mathsf{p_1}$ and $\mathsf{p_2}$ send update messages to q. Now suppose that a tuple $\langle 0, 1\rangle$ is in both $\mathsf{r_1@p_1}$ and $\mathsf{r_1@p_2}$. Then it is correctly in $\mathsf{s_0@q}$. Now suppose that this tuple is deleted from $\mathsf{r_1@p_1}$. Then a deletion message is sent to q, resulting in wrongly deleting the fact from $\mathsf{s_0@q}$.

The problem arises because the tuple $\langle 0, 1\rangle$ had originally two reasons to be in s_0, and only one of the reasons disappeared. To avoid this problem, one could record the *provenance* of the fact $\langle 0, 1\rangle$ in $\mathsf{s_0@q}$. In Section 4.4, we will see a general approach to tracking provenance in our setting, and to using it as basis for performance optimization. For now, the following *Bud* rules is implemented at $\mathsf{p_1}$, $\mathsf{p_2}$ to correctly support the two rules:

[at p1] $\mathsf{s_{0p1}@q(X,Y)}$:- $\mathsf{r_1@p_1(X,Y)}$
[at p2] $\mathsf{s_{0p2}@q(X,Y)}$:- $\mathsf{r_1@p_2(X,Y)}$
[at s] $\mathsf{s_0@q(X,Y)}$:- $\mathsf{s_{0p1}@q(X,Y)}$
[at s] $\mathsf{s_0@q(X,Y)}$:- $\mathsf{s_{0p2}@q(X,Y)}$

Note that relations $\mathsf{s_{0p1}}$ and $\mathsf{s_{0p2}}$ may be either intensional, in which case the view is computed on demand, or extensional, in which case the view is materialized.

E. Non-local rules We consider non-local rules with extensional head. (Non-local rules with intensional head are treated similarly.) An example of such a rule is:

[at p] $\mathsf{r}_0@\mathsf{q}(\overline{\mathsf{X}_0})$:- $\mathsf{r}_1@\mathsf{q}_1(\overline{\mathsf{X}_1}),\ldots,\mathsf{r}_\mathsf{i}@\mathsf{q}_\mathsf{i}(\overline{\mathsf{X}_\mathsf{i}}),\ldots$

with $\mathsf{q}_1 = \ldots = \mathsf{q}_{\mathsf{i}-1} = \mathsf{p}$, $\mathsf{q}_\mathsf{i} = \mathsf{q} \neq \mathsf{p}$, and with each $\overline{\mathsf{X}_\mathsf{j}}$ denoting a tuple of terms. If we consider atoms in the body from left to right, we can process at p the rule until we reach $\mathsf{r}_\mathsf{i}@\mathsf{q}(\overline{\mathsf{X}_\mathsf{i}})$. Peer p does not know how to evaluate this atom, but it knows that the atom is in the realm of q. Therefore, peer p rewrites the rule into two rules, as specified by the formal definition of delegation in *Webdamlog*:

[at p] $\mathsf{mid}@\mathsf{q}(\overline{\mathsf{X}_\mathsf{mid}})$:- $\mathsf{r}_1@\mathsf{p}(\overline{\mathsf{X}_1}),\ldots,\mathsf{r}_{\mathsf{i}-1}@\mathsf{p}(\overline{\mathsf{X}_{\mathsf{i}-1}})$
[at q] $\mathsf{r}_0@\mathsf{q}(\overline{\mathsf{X}_0})$:- $\mathsf{mid}@\mathsf{q}(\overline{\mathsf{X}_\mathsf{mid}})$, $\mathsf{r}_\mathsf{i}@\mathsf{q}(\overline{\mathsf{X}_\mathsf{i}}),\ldots$

where *mid* identifies the message, and notably encodes, (i) the identifier of the original rule, (ii) that the rule was delegated by p to q, and (iii) the split position in the original rule. The tuple $\overline{\mathsf{X}_\mathsf{mid}}$ includes the variables that are needed for the evaluation of the second part of the rule, or for the head. Observe that the first rule (at p) is now local. If the second rule, installed at q, is also local, no further rewriting is needed. Otherwise, a new rewriting happens, again splitting the rule at q, delegating the second part of the rule as appropriate, and so on.

Observe that an evolution of the state of p may result in installing new rules at q, or in removing some delegations. Deletion of a delegation is simply captured by updating the predicate guarding the rule. Insertion of a new delegation modifies the program at q. Note that in *Bud* the program of a peer is fixed, and so adding and removing delegations is a novel feature in *Webdamlog*. Implementing this feature requires modifying the *Bud* program of a peer. This happens during Step 1 of the *Webdamlog* stage.

F. Relation and peer variables Finally, we consider relation and peer variables. In all cases presented so far, *Webdamlog* rules could be compiled statically into *Bud* rules. This is no longer possible in this last case. To see this, consider an atom in the body of a rule. Observe that, if the peer name in this atom is a variable, then the system cannot tell before the variable is instantiated whether the rule is local or not. Also, observe that, if the relation name in this atom is a variable, then the system cannot know whether that relation already exists or not. In general, we cannot compile a *Webdamlog* rule into *Bud* until all peer and relation variables are instantiated.

To illustrate this situation more precisely, consider a rule of the form:

Rule 1
$r_0@p(\overline{X_0})$:- $r_1@p(\$X)$, ...,$\$X@p(\overline{X_i})$,...,

where $r_0@p$ is extensional and $\$X$ is a variable. This particular rule is relatively simple since, no matter how the variable is instantiated, the rule falls into the simple case **B**. However, it is not a *Bud* rule because of the variable relation name $\$X$.

Recall that *Webdamlog* rules are evaluated from left to right, and a constraint is that each relation and peer variable must be bound in a previous atom. (This constraint is imposed by the language.) Therefore, when we reach the atom $\$X@p(\overline{X_i})$, the variable $\$X$ has been instantiated.

To evaluate this rule, we use two *Webdamlog* stages of the peer. In the first stage, we bind $\$X$ with values found by instantiating $r_1@p(\$X)$. Suppose that we find two values for $\$X$, say t_1 and t_2. We wait for the next stage to introduce new rules (there are two new rules in this case). More precisely, new rules are introduced during Step 1 of the *Webdamlog* computation of the next stage. In the example, the following rules are added to the *Bud* program at p:

Rule 2
$r_0@p(\overline{X_0})$:- $t1@p(\overline{X_i})$,...,
$r_0@p(\overline{X_0})$:- $t2@p(\overline{X_i})$,...,

Remark that it is a slight departure from the *Webdamlog* language mentioned in Section 4.2.1. Even if the rule 1 is local, variables force the rule 2 to be evaluated in the next stage. Hence the effects of the rule 1 are postponed. Indeed this can be seen as a peer installing a delegation to itself.

Observe that, even in the absence of delegation, having variable relation and peer names allows the *Webdamlog* engine to produce new rules at run time, possibly leading to the creation of new relations. This is a distinguishing feature of our approach, and is novel to *Webdamlog* and to our implementation.

This example uses a relation name variable. Peer name variables are treated similarly. Observe that having a peer name variable, and instantiating it to thousands of peer names, allows us delegating a rule to thousands of peers. This makes distributing computation very easy from the point of view of the user, but also underscores the need for powerful security mechanisms. The topic of access control is still being investigated ; see [1].

4.3 Optimization of the evaluation

To make the approach feasible, we rely intensively on some known optimization techniques. We briefly mention them next and see how they fit in the *Web-*

damlog picture.

Differential technique

Consider a peer p who has the rule $s@q(x, y)$:- $r@p(x, y)$ with $s@q$ an extensional relation. Suppose that $r@p$ is a very large relation that changes frequently. Each time we visit p, we have to send to q the current version of $r@p$, say a set K_n of tuples. This is a clear waste of communication resources. It is preferable to send the symmetric difference of $r@p$, i.e., send a set of updates Δ with the semantics that $K_n = \Delta(K_{n-1})$, since q already knows K_{n-1}. If $s@q$ is intensional, we face a similar issue; it is preferable to send the new set of delegation rules as Δ rather than sending the entire set.

Seed-based delegation with the differentiation technique

Consider again the rule:

at p: m@q() :- m_1@p($x), m_2@p'($x)

Now suppose that $m_1@p(a_i)$ holds for $i = [1..1000]$. We need to install 1000 rules. However, in this particular case, we can install a single rule at p' and send many facts:

at p': m@q() :- $seed_{r,1,p}$@p'($x), m_2@p'($x)
at p': $seed_{r,1,p}$@p'(a_i). (for each i)

Note that it now becomes natural to use a differential technique to maintain delegation. In particular, if the delegation from p to q does not change, there is no need to send anything. If it does, one needs only to send the delta on $seed_{r,1,p}@p'$. Observe that we have replaced the task of installing and uninstalling delegation rules by that of sending insertion and deletion messages in a persistent extensional (seed) relation that controls a rule.

Query-subquery and delegation

Consider the following example of a rule at Sue where `photos@Sue` is intensional:

```
[at Sue]
photos@Sue($name,$pic) :- photos@Alice($name,$pic)
photos@Sue($name,$pic) :- photos@Bob($name,$pic)
```

This rule says that to find the photos of Sue, one needs to ask the photos of Alice and Bob. The formal semantics says that we install (upload) the rule at Alice and Bob which will result in sending to Sue all the photos of Alice and Bob. However, observe that this has no effect on the state since `photos@Sue` is only intensional. This network traffic may therefore be considered a waste of resources. An optimizer may decide not to prefetch the photos of Alice and Bob to Sue's peer. Now suppose that Sue asks for photos where she's appear:

```
query@Sue($X) :- photos@Sue("Sue",$X)
```

where *query* is an extensional predicate. Now obtaining photos from Alice and Bob changes the state. So the optimizer will install the rules:

```
[at Alice]
photos@Sue("Sue",$pic) :- photos@Alice("Sue",$pic)
[at Bob]
photos@Sue("Sue",$pic) :- photos@Bob("Sue",$pic)
```

Observe that the optimizer performed some form of resolution in the spirit of query-subquery [Vie86, AAHM05a] or rewriting in the Magic Set style [BMSU86] (see also [AHV95]). Indeed, the entire management of delegation can be optimized using these techniques. Note that strictly speaking this may change the semantics of applications: the derivation of some facts may take a little longer than if all the delegations have been installed in advance.

4.4 Optimization for view maintenance

As already mentioned, we are concerned with a highly dynamic context where peer states change, and where peers may come and go. This is a strong departure from datalog-based systems such as *Bud* that assume the set of peers and rules to be fixed. In this section, we discuss how incremental state maintenance is performed efficiently in *Webdamlog* using a novel kind of a *provenance graph*.

4.4.1 Provenance graphs

We use provenance graphs to record the derivations of *Webdamlog* facts and rules, and to capture fine-grained dependencies between facts, rules, and peers. We build on the formalism proposed in [GKT07], where each tuple in the database is annotated with an element of a provenance semiring, and annotations are propagated through query evaluation. Intuitively, semiring

addition corresponds to alternative derivations of a tuple, while semiring multiplication corresponds to joint derivations. Provenance may be represented in the form of a polynomial, or as a graph. We use graphs, because this representation is typically more compact [ADD+11, GKIT10]. Provenance graphs have typed labeled nodes that correspond to provenance tokens and to semiring operations, namely, *or-node* and *and-node*.

Provenance graphs can be used for a number of purposes such as explaining query results or system behavior, and for debugging. Our primary use of provenance is to optimize performance of *Webdamlog* evaluation in presence of deletions. We are also currently investigating the use of provenance graphs for enforcing *access control* and for detecting access control violations.

Provenance graphs have already been considered for datalog evaluation in [GKIT10, ZST+10]. The originality of our approach is as follows:

1. Provenance to optimize deletions via deletion propagation. Systems like Orchestra [GKIT10] already use provenance information for distributed datalog evaluation. However in their case, provenance information is centralized. Like ExSPAN [ZST+10], we maintain provenance information in a distributed manner.

2. In our system, unlike in previous approaches, provenance tokens are assigned to both facts and rules, since rules may be added or removed dynamically. At a given stage, the graph allows identifying the actual fixpoint program that should be run. (Recall that the program changes.) Note that this comes as a complementary technique to optimizations already performed by *Bud*, such as the semi-naive optimization, which assumes that the fixpoint program is fixed.

3. Another distinguishing feature is our use of peer nodes. In a peer p, a large number of rules and facts may come from another peer, say q. This information is recorded, allowing us to react efficiently to q leaving and re-joining the network.

We next illustrate by examples the notion of provenance graph used in our system.

Example 1. Let rn@p be a relation that stores a set of relation names. Consider the following rule that deploys a rule for each relation name in rn@p:

[R01 at p] $X@p(*true*) :- rn@p($X)

We will refer to this rule by its identifier R01. Suppose g_1 and g_2 are in rn@p. Then R01 installs two new rules:

[R01g_1 at p] g_1@p(*true*) :-
[R01g_2 at p] g_2@p(*true*) :-

By a slight abuse of notation, we use rule identifiers to denote the corresponding provenance tokens. Figure 4.1 represents the provenance graph for our example. (Ignore for now the part inside the dashed box).

Rectangular nodes represent the provenance of facts, oval nodes represent the provenance of rules, and pentagons represent peer labels. Circular nodes represent operations of the provenance semiring [GKT07]. The *or-node* represents a disjunction, i.e., alternative ways of deriving the node to which it is connected with an outgoing edge. On the other hand, the *and-node* represents a conjunction: All its in-going edges are needed for the derivation.

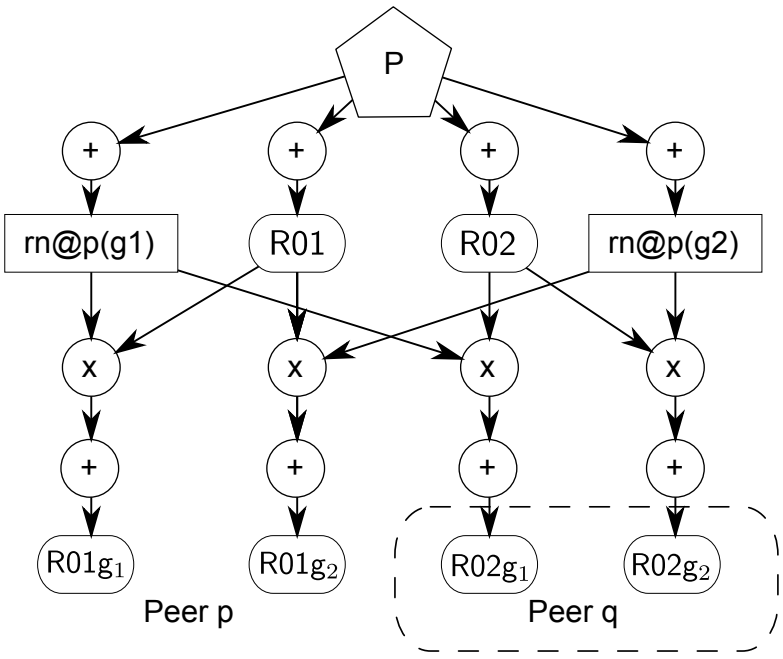


Figure 4.1: Provenance graph

Example 2. In Example 1, the provenance graph is fully stored at peer p. Now consider another rule:

[R02 at p] $X@q(*true*) :- rn@p($X)

Its execution leads to installing at q the following two rules:

[R02g_1 at q] g_1@q(*true*) :-
[R02g_2 at q] g_2@q(*true*) :-

Note that now these rules are outside of p (in q). They correspond to the part of the graph in Figure 4.1 inside a dashed box.

Unlike provenance graph in systems such as ExSPAN [ZST+10] or Orchestra [GKIT10, ADD+11], rules are not labels of edges between nodes storing facts. Instead the rules are recorded as the content of nodes of the graph. Therefore, we can keep their provenance using the same representation as we did for facts. This is necessary because *Webdamlog* rules may be added or deleted at run time either by other rules with variables in peer and relation name, or by delegation from remote peer.

4.4.2 Deletions

When a peer starts a new stage, it may have to process deletion requests that came via the network channels. These deletions are performed just before running the fixpoint. Provenance graphs turn out to be essential for supporting these deletions, because they allow us deleting facts and rules that have been invalidated by the deletions. To do this, when we delete a fact or a rule, we remove its corresponding node from the provenance graph and propagate the deletion throughout the graph. A node is deleted when it loses its last proof of provenance.

4.4.3 Running the fixpoint

The *Bud* engine evaluates the fixpoint using the semi-naive algorithm [AHV95], i.e., *Bud* saturates one stratum after another according to a stratification given by the *dependency graph*. The *dependency graph* is a directed hyper-graph with relations as node and an hyper-edge from relations q_i to p if there is a rule in which all q_i relations appear in the body and p in the head. Since this is classic material, we omit the details, but observe that as rules are added or removed at run-time (as in *Webdamlog*), the program evolves between fixpoint steps (but not within) and so does the dependency graph. The *Webdamlog* engine pushes further the differentiation technique that serves as basis of the semi-naive algorithm. Although in the *Webdamlog* semantic, facts are consumed and possibly re-derived, it would be inefficient to recompute the proof of existence of all the facts at each stage. Between two consecutive stage, each relation keeps a cache of its previous content which could be invalidated by *Webdamlog* if a newly installed rule creates a new dependency for this relation. Note that to some extent, *Bud* is already performing this cache invalidation propagation for facts adding that we adapt to fit *Webdamlog* semantic. This incremental optimization across stages allows us to run the

fixpoint computation only on the relations that may have changed since the previous stage.

The deletion/reinsertion of a single piece of information may have tremendous impact on a peer. Consider for instance peer p that has many rules and facts, the existence of which depends on peer q. Now suppose that q is a smartphone that is often disconnected and re-connected. Peer p must update its knowledge base in response to a change in q's connectivity status, and such updates may be costly. We can use the provenance graph at p, marking the node corresponding to q as *switched off*. As a consequence, a large portion of the provenance graph becomes deactivated, and can be reactivated easily when q reconnects.

This approach may be seen as a generalization of the differential idea used in the semi-naive technique. The semi-naive technique defines the new state I_{new} as $I_{old} \cup \Delta$, so only Δ needs to be sent. Intuitively, the deletion of q should be interpreted as "delete q and record as $I_{old,q}$ all the information that depends on q". An insertion of q now also requests reinstalling $I_{old,q}$.

4.5 Performance evaluation

The goal of the experimental evaluation is to verify that *Webdamlog* programs can be executed efficiently. We show here that rewriting and delegation can be implemented efficiently.

In the experiments, we used synthetically generated data. All experiments were conducted on up to 50 Amazon EC2 micro instances, with 2 *Webdamlog* peers per instance. Micro-instances are virtual machines with two process units, Intel(R) Xeon(R) CPU E5507 @2.27GHz with 613 MB of RAM, running Ubuntu server 12.04 (64-bit). All experiments were executed 4 times with a warm start. We report averages over 4 executions.

The examples are inspired by an implementation of the motivating example described in Section 1, in which friends of Alice and Bob are making a photo album for them as a wedding present. This example is representative of a number of real tasks where many peers collaborate by sharing information. The experiments are designed to capture the salient features of such applications.

The only simplification for the purpose of the experiments is that we assume, to simplify, that each friend keeps his photos on his peer. We work with 3 designated peers representing Alice, Bob and Sue, and with a varying number of peers representing friends of Alice and Bob. Peers alice and bob each contain an extensional relation friends($name). The number of facts in these relations allows controlling the degree of distribution. Each

peer representing a friend of Alice or Bob contains two extensional relations: photos($photoId) and features($photoId,$tag), storing, respectively, the ids of photos and the tags describing the contents of the photos.

4.5.1 Cost of delegation

In this section, the focus is on measuring *Webdamlog* overhead in dealing with delegations. Recall the *Bud* steps performed by each peer at each *Webdamlog* stage, described in Section 4.2.1. We can break down each step into *Webdamlog*-specific and *Bud*-specific tasks as follows:

1. Inputs are collected
   (a) ***Bud*** reads the input from the network and populates its channels.
   (b) ***Webdamlog*** parses the input in channels and updates the dependency graph with new rules. The dependency graph is used to control the rules that are used in the semi-naive evaluation (see Section 4.4.3).
2. Time is frozen
   (a) ***Bud*** invalidates each Δ (used by the semi-naive evaluation) that has to be reevaluated because it corresponds to a relation that may have changed.
   (b) ***Webdamlog*** invalidates Δ according to program updates. Moreover, *Webdamlog* propagates deletions. (Recall that the semi-naive evaluation deals only with tuple additions.)
   (c) ***Bud*** performs semi-naive fixpoint evaluation for all invalidated relations, taking the last Δ for differentiation.
3. Outputs are obtained
   (a) ***Webdamlog*** builds packets of rules and updates to send.
   (b) ***Bud*** sends packets.

We report the running time of *Webdamlog* as the sum of Steps 1b, 2b and 3a, and the running time of *Bud* as the sum of Steps 1a, 2a, 2c and 3b. All running times are expressed in percentage of the total running time, which is measured in seconds. For each experiment, we will see that the running time of *Webdamlog*-specific phases is reasonable compared to the overall running time.

For the experiments in this section, we use *Webdamlog* rules involving *only extensional relations*, both in the head and in the body. We also support rules with intentional relations in the head and in the body. But for such rules, an essential optimization consists in deriving *only the relevant data and delegated rules*. We intend to conduct experiments with such rules when our system supports optimizations in the style of Magic Set.

Non-local rules

In the first experiment, we evaluate the running time of a non-local rule with an extensional head. Rules of this kind lead to delegations. We use the following rule:

```
[at alice]
join@sue($Z) :- rel1@alice($X,$Y), rel2@bob($Y,$Z)
```

This rule computes the join of two relations at distinct peers (rel1@alice and rel2@bob), and then installs the result, projected on the last column, at the third peer (join@sue). Relations rel1@alice and rel2@bob each contain 1 000 tuples that are pairs of integers, with values drawn uniformly at random from the 1 to 100 range. In the next table, we report the total running time of the program at each peer, as well as the break-down of the time into *Bud* and *Webdamlog*.

	Webdamlog	*Bud*	total
alice	10.8%	89.2%	0.10s
bob	4.0%	96.0%	0.87s
sue	0.7%	99.3%	0.02s

The portion of the overall time spent on *Webdamlog* computation on alice is fairly high: 10.8%. This is because that peer's work is essentially to delegate the join to bob. Peer bob spends most of its time computing the join, a *Bud* computation. Peer sue has little to do. As can be seen from these numbers, the overhead of delegation is small.

Relation and peer variables

In the second experiment, we evaluate the execution time of a *Webdamlog* program for the distributed computation of a union. The following rule uses relation and peer variables and executes at peer sue:

```
[at sue]
union@sue($X) :- peers@sue($Y,$Z), $Y@$Z($X)
```

The relation peers@sue contains 12 tuples corresponding to 3 peers (including sue) with 4 relations per peer. Thus, the rule specifies a union of 12 relations. Each relation participating in the union contains 1 000 tuples, each with a single integer column, and with values for the attribute drawn independently at random between 1 and 10 000.

	Webdamlog	*Bud*	total
sue	9.9%	90.1%	1.04s
remote1	1.1%	98.9%	0.04s
remote2	1.3%	98.7%	0.04s

Observe that sue does most of the work, both delegating rules and also computing the union. The *Webdamlog* overhead is 9.9%, which is still reasonable. The running time on remote peers is very small, and the *Webdamlog* portion of the computation is negligible.

QSQ-style optimization

In this experiment, we measure the effectiveness of an optimization that can be viewed as a distributed version of query subquery (QSQ) [Vie86], where only the relevant data are communicated at query time. More precisely, we consider the following view union2 on peer sue, defined as the union of two relations.

```
[at sue]
union2@sue($name,$X) :- photos@alice($name,$X)
union2@sue($name,$X) :- photos@bob($name,$X)
```

Suppose we want to obtain the photos of Charlie, i.e. the tuples in union2 that have the value "Charlie" for first attribute. We vary the number of facts in photos@alice and photo@bob that match the query. We compare the cost of materializing the entire view to answer the query to that of installing only the necessary delegations computed at query time to compute the answer.

Results of this experiment are presented in Figure 4.2. We report the waiting time at sue. As expected, QSQ-style optimization brings important performance improvements (except when almost all facts are selected). This shows its usefulness in such a distributed setting.

4.5.2 Cost of dynamism

This section evaluates the performance of the *Webdamlog* engine in dynamic environments.

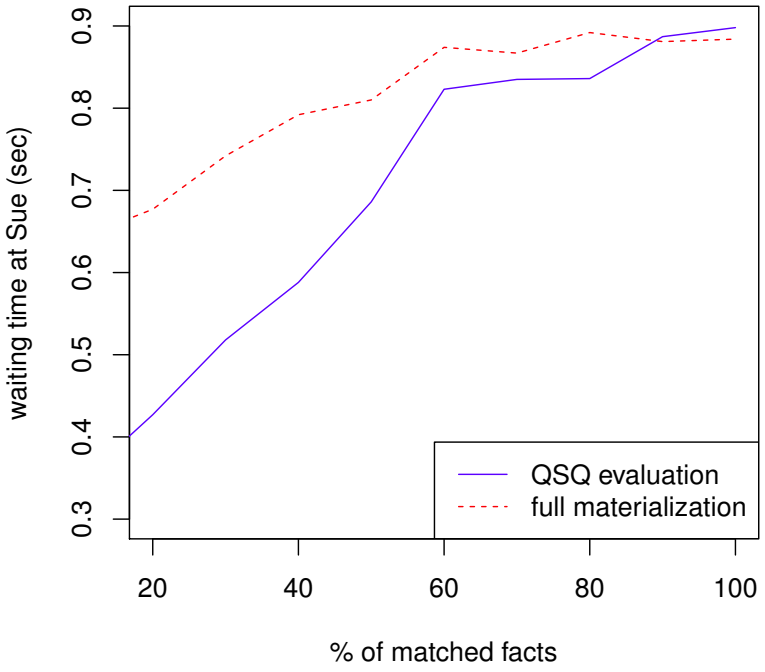


Figure 4.2: Distributed QSQ optimization

For addition of facts and rules, we benefit from semi-naive evaluation in *Bud* and from efficient processing of rule addition in *Webdamlog*. For deletion, we introduced in Section 4.4.1 provenance information in *Webdamlog* computation. We next demonstrate that (i) provenance tracking can be performed at a reasonable cost and (ii) it brings significant improvements when deletions are considered.

Overhead of provenance

In the first experiment, we measure the overhead of this instrumentation. We again use the rules defining allFriends@sue as the union of relations friends at aliceFB and bobFB.

In Figure 4.3 (respectively 4.4), we report the time needed to maintain that union after an update consisting of adding facts to (respectively removing facts from) relations friends@aliceFB and friends@bobFB. We measure the performance of the system as a varying number of facts is added/removed.

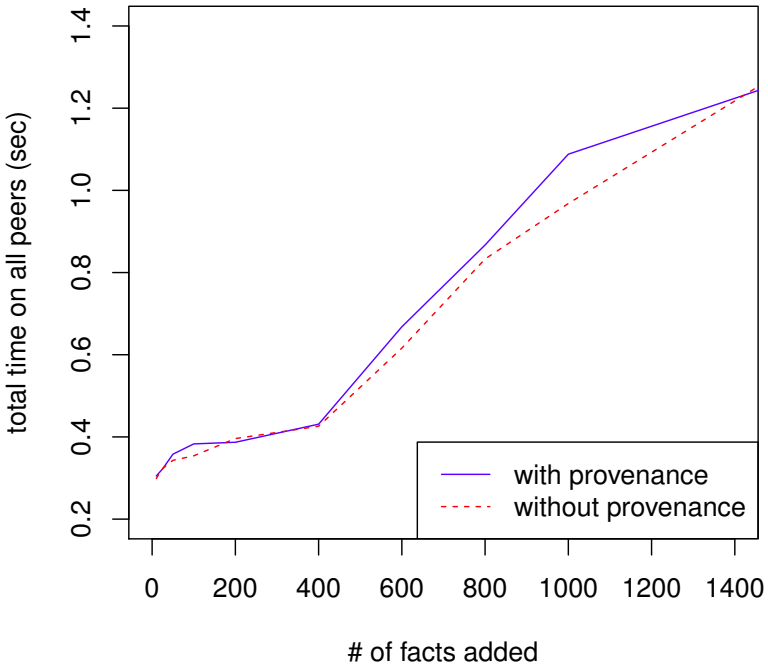


Figure 4.3: Overhead of provenance tracking when adding facts

We report the computing time for *Webdamlog* with and without provenance tracking. We see that the overhead of the instrumentation is small.

Size of the provenance graph

We also measure the size of the provenance graph as the number of dependencies increases. For that, we constructed an example with a large number of facts (1 000 000 in total) so that we can considerably grow the dependencies between facts (each fact will eventually have a very large number of proofs).

We use 10 peers ($p_1..p_{10}$), 100 relations on each of them ($r_1..r_{100}$ of arity 1) with 1 000 facts in each relation (containing an integer between 1 to 10 000). Each rule on peer i is of the form:

$r_{j'}@p_k(X)$:- $r_j@p_i(X)$

i.e., the rule has a unique relation in the body. (The way these rules are selected is irrelevant.) We increase the number of rules on each peer from 1 to

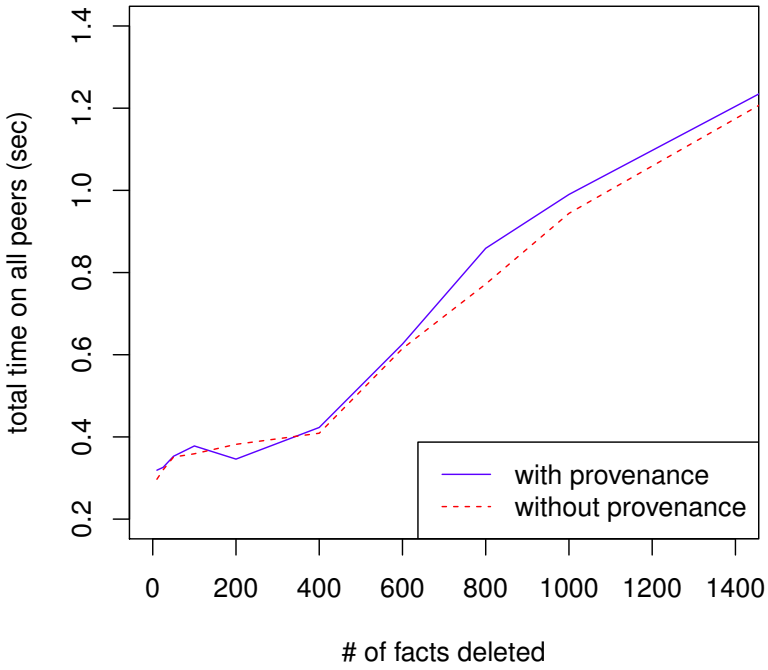


Figure 4.4: Overhead of provenance tracking when deleting facts

100 000. (Each of the 100 relations of this peer is connected to 1 000 relations of the 10 peers.) Thus, the total number of rules in the system varies from 10 to a million. At this extreme, the content of each relation is copied into each relation in the system.

In Figure 4.5, we report the total size of the provenance graph. Observe that the provenance graph is split equally across 10 peers, and so each peer stores one tenth of the total size. We see here that the size of the provenance graph grows linearly in the size of the program. Observe that, in this already complex case, the size of the provenance graph is still reasonable (about 44MB per peer), and is notably small enough to be kept in main memory.

Performance of deletion propagation

In this experiment, we demonstrate the performance gains brought by the use of the provenance graph for deletion propagation. For this, we use a more complex setting. We have 10 peers, each containing a source relation

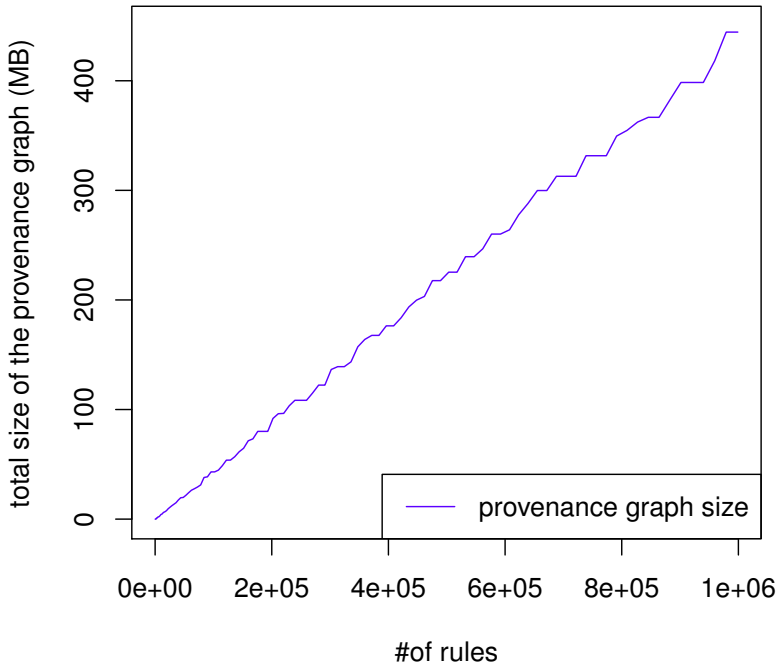


Figure 4.5: Size of the provenance graph compared to the size of the program

(source@p_i, for i from 1 to 10) with 1 000 facts in each. Then we have 6 layers of 10 peers, each containing an intermediate relation (inter@p_{ij}, for i from 1 to 10, and j from 1 to 6). Finally, we have a unique target relation that gathers all facts. Each fact in a source relation propagates to 3 relations in the first layer. Each fact in layer $j < 6$ propagates to 3 relations in layer $j+1$. Each fact in layer 6 propagates to the target relation.

Figure 4.6 compares the time it takes to update the target relation (i) by propagating deletions (propagation) and (ii) by fully recomputing the peer states (recomputation). We vary the number of deleted facts between 5 and 1 000 facts for each relation source@p_i. We observe that even in such a case, with rather complex dependencies, deletion can be supported efficiently thanks to the provenance graph.

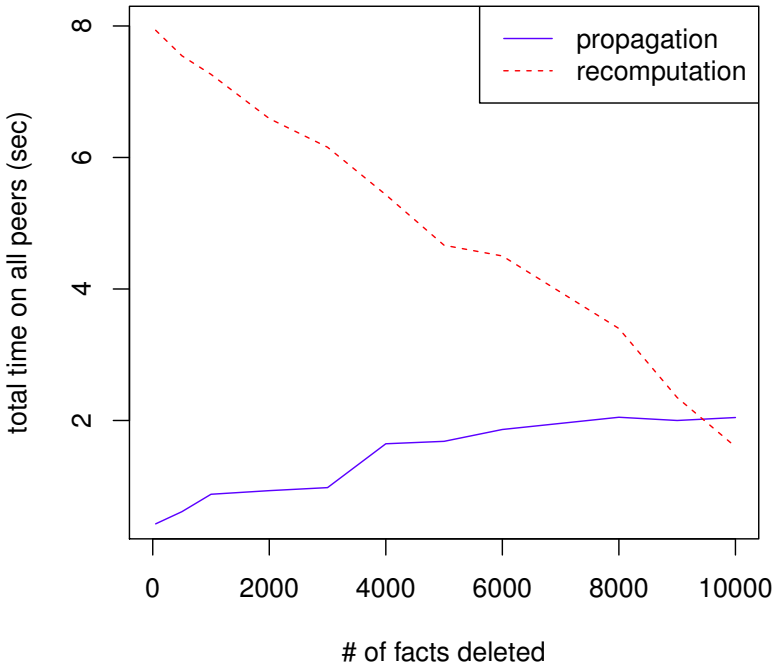


Figure 4.6: Deletion propagation vs. recomputation

Distribution and evolution

Finally, we measure the performance of our system for the following rule using 100 *Webdamlog* peers on 50 Amazon micro-instances with two peer per instance:

```
[rule at sue]
album@sue($photo,$peer) :-
   allFriends@sue($peer),
   photos@$peer($photo),
   features@$peer($photo,alice),
   features@$peer($photo,bob)
```

This rule delegates processing to multiple peers, with these peers determined by the content of the relation allFriends@sue. We measure the cost of maintaining the photo album when between 5 and 100 sources are deleted.

This experiment shows the performance of *Webdamlog* under such updates. We compare two strategies:

1. Our strategy that propagates changes using the provenance graph without fixpoint computation.

2. A baseline strategy that recomputes the new set of rules, reinitializes the peer with these rules, and restarts the *Bud* fixpoint computation from scratch.

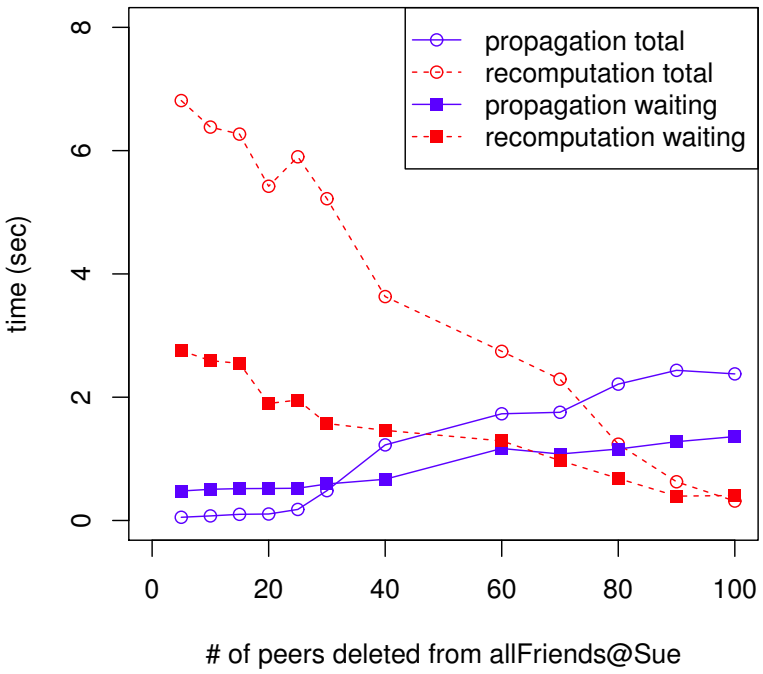


Figure 4.7: *Webdamlog* evaluation during program update

In Figure 4.7, we report two measurements for each strategy: (i) total time and (ii) waiting time at sue, between the moment sue requests the update and the end of the computation. We observe that, in terms of waiting time at sue, deletion propagation significantly outperforms full recomputation when no more than 60 peers are deleted. A similar trend holds for total time. If sue decides to remove the majority of her friends, full recomputation performs better, as expected.

Chapter 5

Architecture of a *Webdamlog* peer

Information management on the Internet relies on a wide variety of systems, each specialized for a particular task. A user's data and favorite applications are often distributed, making the management of personal data and knowledge (i.e., programs) a major challenge. Consider *Joe*, a typical Web user who has a blog on Wordpress.com, a Facebook account, a Dropbox account, and also stores data on his smartphone and laptop. *Joe* is a movie fan and he wants to post on his blog a review of the last movie he watched. He also wishes to advertise his review to his Facebook friends and to include a link to his Dropbox folder where the movie has been uploaded. This is a cumbersome task to carry out manually, yet automating it, for example by writing a script, is far beyond the skills of most Web users.

Some systems attempt to provide integrated services to support such needs. For instance, Facebook provides a wrapper service to integrate Dropbox accounts and blogs. However, such services are often limited in the functionality they support. Also, by delegating such services to systems like Facebook, a user needs to trust more and more of his information to one particular system. Our goal is to enable the user to easily specify distributed data management tasks *in place* (i.e., without centralizing his data to a single provider), while allowing him to keep full control over his own data as presented in [2]. Our system is not a replacement for Facebook, or any centralized system, but it allows users to launch their customized peers on their machines with their own personal data, and to collaborate using Web services. Our contribution in designing an architecture around a *Webdamlog* rule engine is presented in [1, 4, 5].

This chapter describe a *Webdamlog* peer that embeds the *Webdamlog* engine and uses wrappers to other systems. The focus is on theses wrappers

that allow a *Webdamlog* peer to integrate data of non-*Webdamlog* peers.

Organization The chapter is organized as follows. We introduce the full peer architecture around the *Webdamlog* engine in Section 5.1. Section 5.2 discusses the integration of non-*Webdamlog* peers using wrappers. Finally, in Section 5.3, we present the demonstration of an example of application, namely Wepic.

5.1 Peer architecture

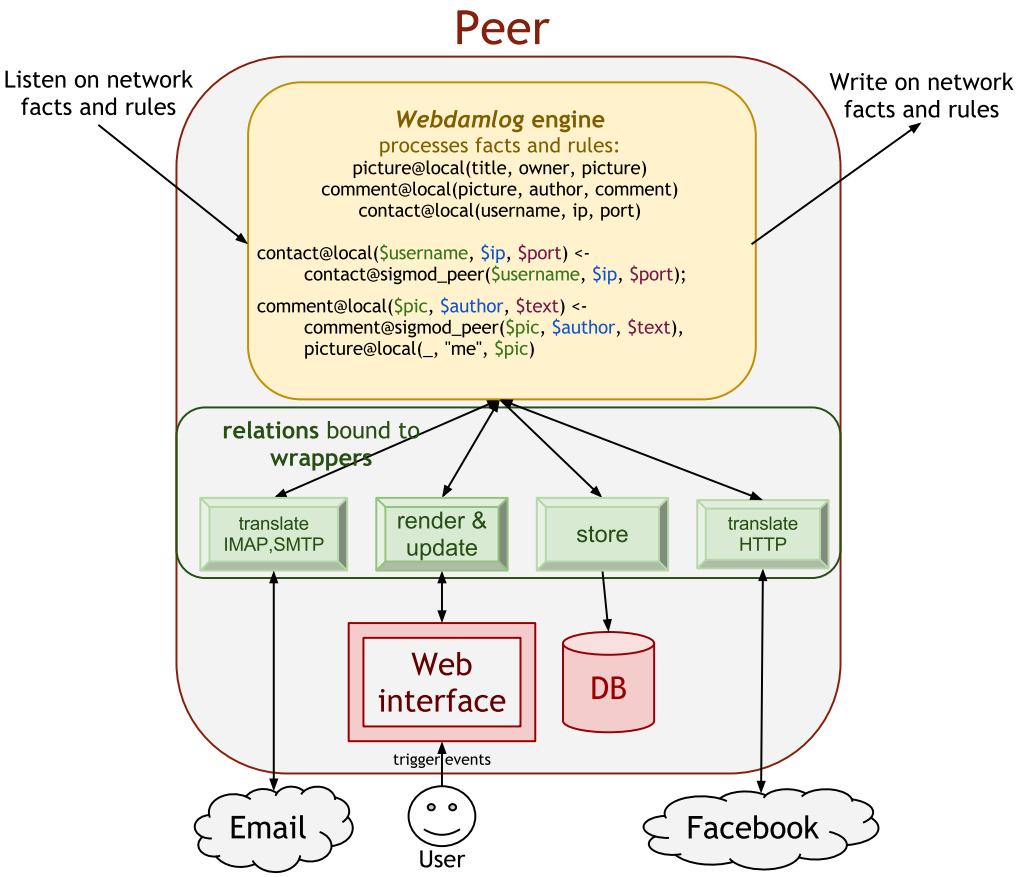


Figure 5.1: *Webdamlog* peer architecture

In this section, we describe a particular implementation of a *Webdamlog* peer. Figure 5.1 gives an overview of the connection between the *Webdamlog*

engine and other modules of a full peer. The *Webdamlog* engine at the heart of the system receives facts and rules from remote *Webdamlog* engines in *Webdamlog* format. A set of wrappers bound to selected relations in the *Webdamlog* engine can read/write in these relations. For instance, the user has a user-friendly view of the *Webdamlog* knowledge via a Web interface. He can trigger updates of the relations or rules. A wrapper called *renderer* is needed to display content of the *Webdamlog* engine relations in HTML and keep the display synchronized while the *Webdamlog* engine is modifying its relations. Additional wrappers provide communication with non-*Webdamlog* peers such as Facebook ; ability to send emails ; and persistent storage in databases.

The *Webdamlog* peer consists of a set of programs, that are deployed and linked to a *Webdamlog* engine. A short description of each of them is given next and the corresponding wrappers are detailed in Section 5.2.

Web server The user interface is served by *Thin*, a lightweight HTTP server. Contrary to usual thread-based Web server such as the popular *Apache* that forks to create one thread for each requests, *Thin* is an event-driven server. In the case of Web server, events are HTTP requests issued by the user, that are enqueued and dispatched according to the availability of resources. Event-driven mechanisms are at the core of *Webdamlog* peers and are detailed in Section 5.1.1.

Persistent storage The current implementation supports three different storage software. *Gdbm* a lightweight key-value store, *SQLite 3* a light-weight relational database and *PostgreSQL* a sophisticated relational database. For the most common usage *Gdbm* and *SQLite* are the best choices since they are fast and require no configuration. However the more complex *PostgreSQL* may be useful, for instance to keep an history of the past stages of the *Webdamlog* engine.

Contrary to the *Webdamlog* engine alone, a *Webdamlog* peer may receive events from different sources at the same time e.g. the user can update a relation while the *Webdamlog* engine receives packets on its channels. Concurrency issues are considered in Section 5.1.1.

The *Webdamlog* engine uses only relations as data structures, whereas wrappers may manipulate other kinds of data: trees, large binary object, etc. The specification of an API for designing wrappers integrating *Webdamlog* relations safely is discussed in Section 5.1.2.

5.1.1 Event-driven system

As shown in Section 5.1, a *Webdamlog* peer receives events from remote *Webdamlog* peers, users and wrappers. All these events happen concurrently. But according to a *Webdamlog* stage split in *Bud* three steps described in Section 4.2.1, it is required that during the firing of a *Webdamlog* stage, messages are used at Step 1 following the *Webdamlog* language semantic ; no updates occur at Step 2 until a fixpoint is reached ; and all messages must be sent to each peers as single packet. Hence the architecture of a *Webdamlog* peer must guarantee atomicity of *Webdamlog* engine stages.

The naive implementation of a *Webdamlog* peer would be to launch the *Webdamlog* engine and all other wrappers as separated processes and use inter-process communications and locks to guarantee a safe access to resources. However in this case the fairness of access to resources would be left to the operating system scheduler. This architecture leads to poor performances due to the overhead of context switching and possibly deadlocks. For example, let us consider a simple *Webdamlog* engine receiving successively many messages from other *Webdamlog* engines and a Facebook wrapper, launched as two different processes on the same peer. Suppose that the Facebook wrapper starts a request that takes a long time to be processed. Each time the Facebook thread is dispatched to the CPU by the scheduler, it will be blocked until the request answer has been received, in which case a lot of time spent waiting for IO events are wasted.

Therefore the *Webdamlog* peer adopts an efficient event handling service following the reactor design pattern detailed in [Sch95]. The general idea is to have only one process handling all events to dispatch according to resource availability. As depicted in Figure 5.2, the reactor is the only process dealing with input/output interruption.

The reactor runs an event loop listening for all events registered, e.g. in Figure 5.2, HTTP requests, emails and UDP network packets. Each event is associated to some code to execute that is called by the dispatcher if the resource is available. For instance, the reactor may receive a *Webdamlog* packet from a remote peer via an UDP port that is associated with the *Webdamlog* engine. The reactor knows that to handle this event it should dispatch the packet to the *Webdamlog* engine that will fire a new stage. The way the dispatcher works is out of the scope of this thesis but it is fully customizable for the *Webdamlog* peer instead of the multi-process solution that lets the OS takes all the decision. Note that the fact that the reactor is the only process handling input/output, does not imply that the system is single threaded. For instance, long running tasks that are not updating relations directly can be delegated to threads. E.g. the waiting time when

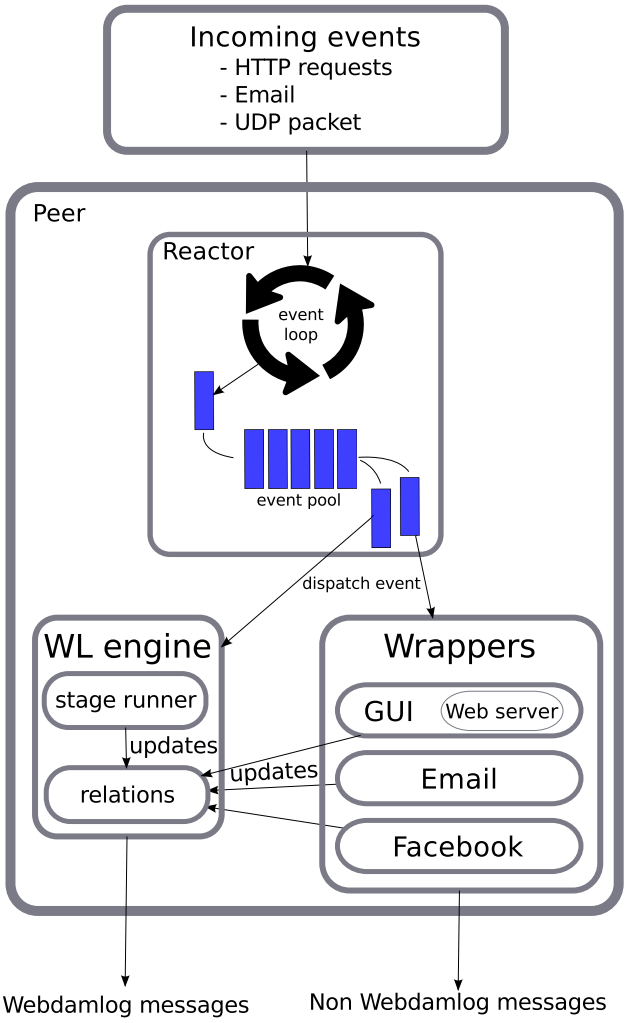


Figure 5.2: Event handling in *Webdamlog* peer

the Facebook wrapper sends a requests is delegated to a thread.

This reactor design pattern gives a clear specification for the modules to be used in a *Webdamlog* peer. Each module must specify event listeners and the handlers i.e. the codes to be executed as callback methods invoked by the reactor. In the *Webdamlog* peer implementation, a Ruby implementation of the reactor pattern named Event-machine [Eve13] is used.

5.1.2 Module interactions

As shown in Figure 5.2, *Webdamlog* relations can be read/updated by different wrappers as well as by the *Webdamlog* engine. The *Webdamlog* peer is designed

such that the *Webdamlog* engine does not directly interact with wrappers. *Webdamlog* relations can be modified by wrappers but not during *Webdamlog* stages. The concurrent accesses to the relations is supported thanks to the reactor system.

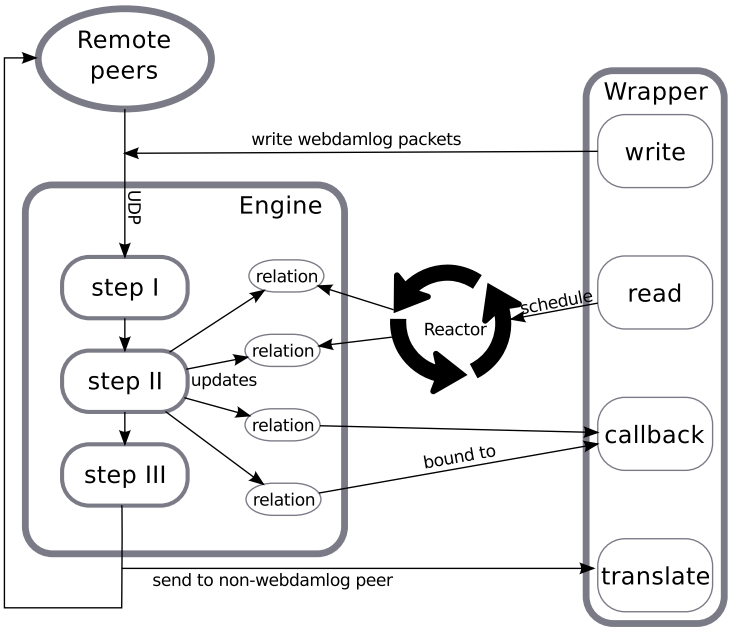


Figure 5.3: Wrapper running in parallel with *Webdamlog* stage

Figure 5.3, gives a representation of the interface between a wrapper and the *Webdamlog* engine. Two methods, namely `read` and `write` provide the communications from the wrapper to the *Webdamlog* peer. The methods `read` and `write` trigger the firing of a new *Webdamlog* stage via events scheduled in the reactor. The others two, namely `callback` and `translate`, correspond to the propagation of changes occurred in the engine as side-effects handled by the wrapper.

In Figure 5.3, the *Webdamlog* engine is running a stage in 3 steps. When running a stage, new packets may be emitted to remote peers and relation updates may occur during Step 2.

Read The wrappers use the asynchronous read method of the *Webdamlog* engine. This read method is a read order in the form of an event scheduled in the reactor queue. It takes a list of relations to read at the same stage as argument, and returns their content. When this event is triggered, it forces the engine to fire a stage and return the actual content of relations at the end of Step 2. This prevents the *Webdamlog*

engine from updating relations while reading. Note that a read order, could be seen as sending an empty packet to the peer and returning the projection of all the relations asked. However, if other packets are pending on the channel, they will be processed, and the content returned will be updated accordingly.

Write Wrapper sends a *Webdamlog* formatted packet on the regular UDP port of the *Webdamlog* engine. It writes facts and rules and serializes them to be processed as other packets.

Translate From the *Webdamlog* engine point of view, a wrapper may be seen as a remote *Webdamlog* peer, therefore the wrapper may receive *Webdamlog* packets of facts and rules. The wrapper simulates a *Webdamlog* peer hence it accepts a limited type of facts and rules. The schema of facts and rules accepted defines the API of the wrapper. Therefore the translate method filters out non-conform *Webdamlog* facts and rules. Then it invokes the Ruby code to execute in response to *Webdamlog* packets.

Callback updates There must be as many callback methods as relations that are bound with the wrapper. Each callback method takes as arguments a list of facts. The callback method is invoked by the *Webdamlog* engine as soon as changes occur in the relation. The callback methods receive the delta of facts that defines the facts that are added/removed at the current stage. These methods must be used to propagate internal updates to the external program that the wrapper handles.

5.2 Wrappers

As shown in Figure 5.4, a wrapper in a *Webdamlog* peer is a code that provides an interface between, one or several *Webdamlog* engine relations and, a non-*Webdamlog* peer. Thus the *Webdamlog* peer speaks to the non-*Webdamlog* peers using the *Webdamlog* relations bound to the wrapper. The interface corresponds to event listeners as detailed in Section 5.1.1. The *Webdamlog* peer implementation follows the Web standards and the programming concepts of the Rails framework [Rai13].

The Ruby object must implement at least the four methods detailed in Figure 5.3: read, write, translate and callback updates. The body of the wrapper is an object-relational mapper (ORM) following the Active Model interface detailed in [Act13]. Active model is now the standard API to build custom ORM in Rails.

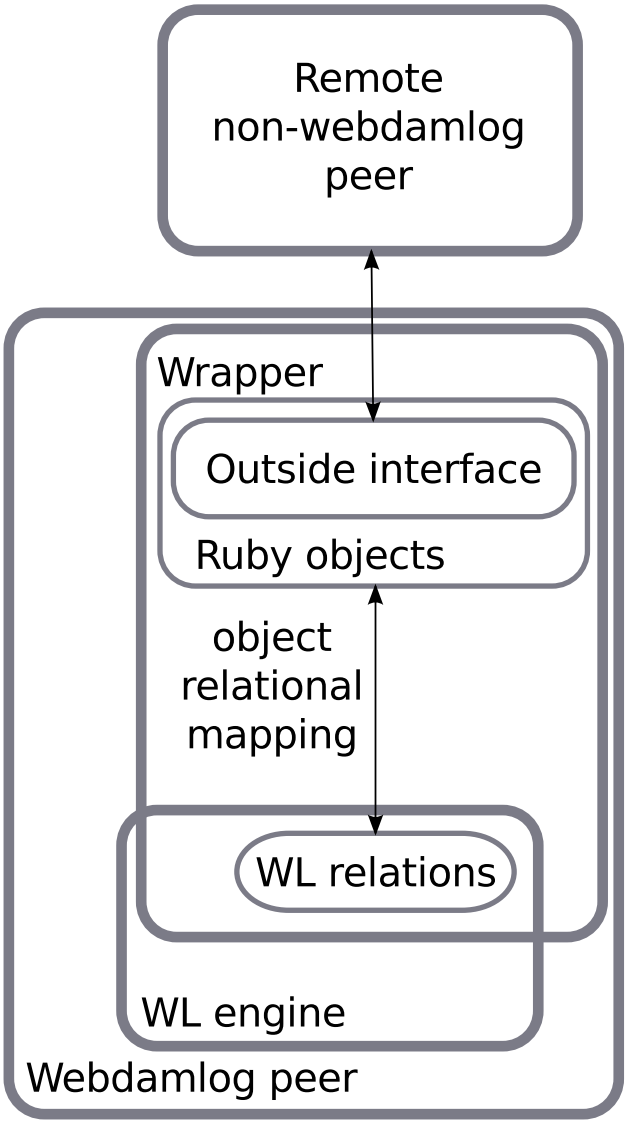


Figure 5.4: Wrapper architecture in a *Webdamlog* peer

Storage wrapper At the end of each stage, the callback methods save the content of relations bound to this wrapper into the linked database along with the program. It also allows the *Webdamlog* peer to reload the relations and the program to reboot the peer from a previous state that has been saved. If the database provides some journaling mechanism, it also allows to restart the peer from a previous state in case of crash. Note that this wrapper is never triggering events to the *Webdamlog* peer, contrary to the next wrappers. Implementation of this wrapper follows the Active Record pattern [Fow02], a

standard for ORM with persistent storage.

Translation wrappers These wrappers simulate a remote *Webdamlog* peer synchronizing some relations that represent a particular view of the data of the remote service. For example, the Facebook wrapper used in Section 5.2 simulates a Facebook *Webdamlog* peer that represents the URL: `www.facebook.com`. The Facebook wrapper provides facilities for authentication on Facebook. Once a given *Webdamlog* user has given his Facebook credentials, the wrapper simulates a peer (say ÉmilienFB). According to the features supported by the wrappers, it provides an abstract view of Émilien's Facebook data as a set a *Webdamlog* relations. E.g. in Section 5.3 the wrapper provides two relations:

friends@ÉmilienFB($userID, $friendName)
pictures@ÉmilienFB($picID, $owner, $URL)

that are the representation in *Webdamlog* relations of the list of friends and the list of pictures of Émilien's account on Facebook. For Facebook, the wrapper needs to send the http query with the right credential to retrieve the list of pictures in JSON. Then it translates this JSON data into a *Webdamlog* collection. Conversely it does the opposite to send pictures. Note that on the *Webdamlog* peer Émilien, the relation friends@ÉmilienFB and pictures@ÉmilienFB receive updates from Facebook during Step 1 of a *Webdamlog* stage and send updates to Facebook during Step 3. However during Step 2, the relations of ÉmilienFB are processed as if there were local to Émilien therefore the rules containing such atoms is delegated but processed as a if they were local. Remark that the name ÉmilienFB uniquely identifies the peer as the Facebook account of Émilien is generated by the given Facebook wrapper based on the Facebook credential of Émilien. Hence another peer with the same Facebook wrapper and the Facebook credential of Émilien would also process ÉmilienFB atoms locally.

User interaction In the implementation, the GUI is rendered by a light weight web server, namely *Thin*. The GUI wrapper translates *Webdamlog* engine collections into HTML+JavaScript+JSON code. The user triggers the JavaScript function that calls the Ruby methods of the wrapper to interact with the *Webdamlog* engine. All the interactions that is reading or updating facts or rules occurs outside a *Webdamlog* stage. This is guaranteed by the event-machine described in Section 5.1.1. Therefore the updates of the user are stacked up and processed the next time the *Webdamlog* engine fires together with the messages received from remote peers. The GUI is built on the Rails standard following the Model-View-Controller framework. Each *Webdamlog*

relation is represented by models i.e. Ruby objects implementing the Active Model API. All requests from the user are RESTful actions processed by controllers.

5.3 Demonstration

This demonstration [4] has been presented at SIGMOD 2013. The Wepic application is a distributed picture manager. The Wepic application is specified using simple rules written in *Webdamlog* described in Chapter 3 and uses a *Webdamlog* engine described in Chapter 4.

A central issue in such a setting is the ease with which a casual user can write *Webdamlog* rules. We conducted a user study described in Chapter 6, showing that users are able to both understand and write simple *Webdamlog* programs after a short tutorial as shown in Section 6.1. The Wepic demonstration shows the simplicity of the *Webdamlog* programs need to designed standard applications that handle personal data.

SIGMOD attendees could use Wepic to share, download, rate and annotate pictures taken at the conference. Attendees could launch their own Wepic peer and interact with the application via a Web GUI. They first inspected the basic *Webdamlog* rules of the provided application and then were invited to customize the application by modifying or adding rules.

5.3.1 Wepic application

Wepic behavior is driven by a small set of *Webdamlog* rules that we discuss further. In addition, the application uses two standard wrappers, one for Facebook, and one for email communications. The *Webdamlog* system also provides a graphical user interface (GUI), which has been customized to provide a user interface for Wepic. A Wepic peer can:

1. Upload a picture from a file or a URL;
2. View pictures provided by a particular attendee;
3. Transfer pictures:
   (a) send them by email to the SIGMOD group on Facebook, or to another Wepic peer,
   (b) get pictures from another Wepic peer or from the SIGMOD group on Facebook;

4. Annotate pictures with ratings, comments or name tags (names of attendees appearing in the picture);

5. Select and rank photos based on their annotations.



Figure 5.5: A screenshot of the Wepic user interface.

The GUI is a particular kind of wrapper relying on an internal web server as detailed in Section 5.2. In this case the wrapper produces web pages in ERB [Rub], HTML [W3C13] and Javascript [Byn13]. The user event triggered in the GUI are transformed into adding or deleting facts or rules in the *Webdamlog* engine.

We now illustrate how some of these functionalities are implemented with *Webdamlog* rules. To view pictures uploaded by a particular SIGMOD attendee, we use a relation selectedAttendees that contains one fact for each currently highlighted attendee (see right-hand side column in Figure 5.5 sigmod_peer is selected). We also use a derived relation pictures, that is the view of all the pictures of a particular attendee. To obtain the pictures of all selected attendees, we use the rule:

```
attendeePictures@Jules($id, $name, $owner, $data) :-
        selectedAttendee@Jules($attendee),
        pictures@$attendee($id, $name, $owner, $data)
```

Note that this rule uses delegation, a feature novel to *Webdamlog*, to retrieve the contents of relation pictures of each attendee. The result of executing this rule is shown in the frame named sigmod_peer's pictures in the middle of Figure 5.5.

To transfer pictures between peers, we assume that each attendee specifies some preferred communication protocols in relation communicate, stating, e.g., whether he prefers to receive pictures by email, by posting on Facebook, or directly in his Wepic peer. The following rule is executed when Jules sends some pictures to some attendees:

```
$protocol@$attendee($attendee, $name, $id, $owner) :-
        selectedAttendee@Jules($attendee),
        communicate@$attendee($protocol),
        selectedPictures@Jules($name, $id, $owner)
```

Rules of this kind, and other rules implementing the basic functionality of Wepic, are available in the Wepic application, for inspection and customization through the user interface. An example of interface to customize these rules is shown in Figure 5.6, the peer sigmod_peer has just received a delegation asking to send all its contact to Julia. It also have installed three previous delegation in its program.

Delegation and access control By using delegation a user may write a rule and ask another peer to process it remotely. Consider again the previous rule:

```
attendeePictures@Jules($id, $name, $owner, $data) :-
        selectedAttendee@Jules($attendee),
        pictures@$attendee($id, $name, $owner, $data)
```

Suppose we have the facts:

```
selectedAttendee@Jules("Émilien")
```

The evaluation of the rule leads to delegating the following rule to Émilien:

```
attendeePictures@Jules($id, $name, $owner, $data) :-
        pictures@Émilien($id, $name, $owner, $data)
```

This rule requires the peer Émilien to send all the facts in his relation pictures to Jules. This is a simple case of delegation, which can be controlled by *inferring access* from the specifications described above. However delegated



Figure 5.6: The interface to a *Webdamlog* program running Wepic.

rules can be more complex, and general methods for effectively controlling delegation are a topic of on-going investigation considered in Section 5.3.3.

5.3.2 Demonstration Scenario

We now describe the general proceedings of the demonstration. The goal is to share pictures taken during the SIGMOD conference. Émilien and Jules are attendees of the conference. They have used Wepic to install locally on their laptops a collection of pictures. They demonstrate how to use Wepic with the native functionalities described in Section 5.3.1 and how to customize the application. User at the conference are also allowed to run their own Wepic peer to explore the system. This scenario demonstrates the various aspects of *Webdamlog*, notably distribution, delegation and control of delegation.

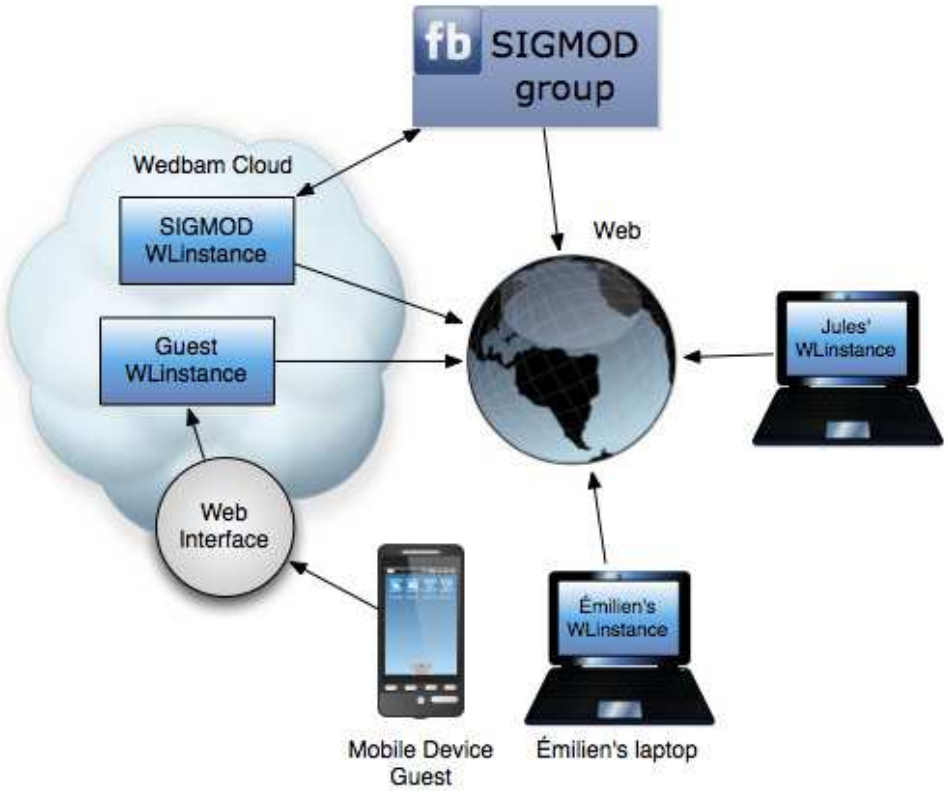


Figure 5.7: The distribution of peers in the network.

Setup In the beginning of the demo, the peers are distributed as shown in Figure 5.7. Three peers are established: one on each of the laptops of Émilien

and Jules, connected via a local network, and a third, the SIGMOD peer, hosted on the *Webdam cloud*. To simplify the presentation, it is assumed that Émilien and Jules have the same bootstrap program so they organize their data and behave similarly. They both store their personal photos in pictures@Émilien and pictures@Jules on their respective Wepic peers. Both have Facebook accounts and are members of the SigmodFB group, the official Facebook group of the conference. Finally, both users are subscribed to the SIGMOD peer, that stores the list of registered Wepic users.

Peer discovery To enter in the network, one peer should know at least one other peer already linked with some others. In this particular application Wepic, we setup the initial knowledge base of new peers with the public URL of the SIGMOD peer. For example the Émilien peer is initialized with this fact:

attendee@Émilien("Émilien", "81.205.87.245:60")

and subscribes to the SIGMOD picture network by sending its address thanks to the following rule included in the bootstrap program:

attendee@SIGMOD("Émilien", $URL) :- attendee@Émilien("Émilien", $URL)

In the SIGMOD peer, the program is setup with the following rule that allows the SIGMOD peer to act as a Hub that broadcasts all the attendees known by SIGMOD to all attendee peers:

```
attendee@$att($member, $URL) :- attendee@SIGMOD($att, _),
                                attendee@SIGMOD($member, $URL)
```

This simple strategy of peer discovery is part of the Wepic program but one can write other strategies.

During the demonstration, attendees can also start their own peers with their personal photos. Since it would be too long to install a Wepic peer on the laptop or smartphone of an attendees, we propose an alternative solution. The attendee can connect to the Web interface on the *Webdam cloud* to launch their own dedicated peer with the same program as Émilien and Jules. Then they can upload their photos and modify their program, as we do on the laptop-based peers.

We start the demonstration by quickly going over the setup while attendees are starting their new peer. They observe that their peer is automatically receiving the list of attendees logged in. Then they can interact with Wepic in the following ways.

Interaction via Facebook To illustrate the interaction between a Wepic peer and other Web services, we use a Facebook wrapper. For instance, the following rule is used by the SIGMOD peer to automatically publish, on the Facebook group of SIGMOD, the pictures belonging to SIGMOD attendees who have authorized this action:

```
pictures@SigmodFB($id, $name, $owner, $data) :-
        pictures@SIGMOD($id, $name, $owner, $data),
        authorized@$owner("Facebook", $id, $owner)
```

Conversely, the SIGMOD peer automatically retrieves the pictures with their comments and tags from the Facebook group and publish them to SIGMOD peer. Note that the system thus allows any Wepic user to see or publish (via Wepic) pictures in SigmodFB even without having a Facebook account. Likewise it allows any *Webdamlog* peer even if they don't have a Facebook wrapper to publish on Facebook. A user only needs to appropriately populate his authorized relation to control Facebook publication. This is typical case where delegation provides a functionality without the need to install the wrapper by itself.

We explain the *Webdamlog* rules that implements these interactions to audience members. And then we show that a photo uploaded by Émilien into his local relation pictures@Émilien is instantly published to pictures@SIGMOD, and then propagated to pictures@SigmodFB.

Customizing rules The main advantage of a peer-to-peer system such as *Webdamlog* is the ability to customize a peer's behavior. Therefore the most novel trait of Wepic is that it lets the user customize existing rules and add his own rules. For example, a user who is interested only in the pictures that have a rating of 5 would customize the rule of the application as follows:

```
attendeePictures@Jules($id, $name, $owner, $data) :-
        selectedAttendee@Jules($attendee),
        pictures@($id, $name, $owner, $data),
        rate@$owner($id, 5)
```

Redefining this rule changes the contents of the frame *Attendee pictures* in Figure 5.5, which has been demonstrated. Then they are free to customize the rule further, retrieving, e.g., only pictures that were taken by a certain SIGMOD attendee, or in which only certain attendees appear using some meta-data tags added by Facebook and retrieve at SIGMOD peer.

Illustration of the control of delegation To illustrate the control of delegation, Émilien attempts to install a rule at Jules' peer. We show that the system requires the approval of Jules before installing the rule, and that the program of Jules is changed once the approval is granted and the rule is installed.

5.3.3 Access control

We briefly describe an important issue that is still not supported by *Webdamlog*, namely access control, see ongoing-work [1].

Since many *Webdamlog* applications manage personal or social data, access to sensitive information must be carefully controlled. Access control in *Webdamlog* is particularly challenging because of the distributed nature of computation and the ability of peers to delegate rules to other peers.

The demonstration of Wepic provides a simplified model for control of delegation, in which each delegation sent by an untrusted peer is pending in a queue until the user explicitly accepts it via the Web interface. A notification of a pending delegation can be seen at the bootom of Figure 5.6, where Julia is sending a rule to sigmod_peer. By default, all peers except the SIGMOD peer are considered untrusted.

A complete access control model for *Webdamlog* is under investigation see [1] and will not be discussed in this thesis. In that model, access to stored or derived relations is controlled by a novel combination of both discretionary methods (in which users have the power to grant rights to data they own) and mandatory methods (in which access rights are derived according to system-wide conventions). Users directly specify the accessibility of extensional relations stored that they own. For derived relations (i.e. views), a user may rely on a default access control policy that is derived automatically from the provenance of the base relations. Alternatively, a user may override this policy in order to grant access to views, effectively "declassifying" some data. This flexible model subsumes the view-based access control of the standard SQL authorization model.

Chapter 6

User Study

We conducted a limited user study to verify that the *Webdamlog* language can be understood and written by non-programmers. We wanted to highlight how some tasks that would be long and complex to write in standard programming language (Java, Python, ...), can be written in *Webdamlog* by regular users. Clearly it would be interesting to perform more thorough user study in particular to help design the user interface.

In this chapter, we present first the tutorial given to the user, then the test. Finally, we present a comment of the results that were obtained.

6.1 *Webdamlog* tutorial

The original tutorial was a set of slides that is reformatted next. A teacher explained the slides to the users in a brief 20 minute lesson.

Terminology

- A *relation* is a database table.
- A *fact* is an entry in a relation.
- A *relation* has a schema, describing attributes of each fact that belongs to it.
- Relations reside at *peers*.

Examples:

$$\underbrace{birthdays@facebook : name, date}_{schema}$$

$$\underbrace{birthdays}_{relation} @ \underbrace{facebook}_{peer}(\underbrace{"Ann", "6/15"}_{fact})$$

Rules (I)

- Suppose that there is a relation photos@picasa, with schema:

  photos@picasa: fileName, content

- Suppose that photos@picasa contains the facts:

  $$photos@picasa(\text{"}image1.jpg\text{"}, \text{"....."})$$
  $$photos@picasa(\text{"}image2.jpg\text{"}, \text{"....."})$$

- We can copy facts from photos@picasa into photos@myLaptop (with the same schema) using the following *rule*:

  $$photos@myLaptop(\text{"}image1.jpg\text{"}, \overbrace{\$X}^{variable}) \text{:-}$$
  $$photos@picasa(\text{"}image1.jpg\text{"}, \underbrace{\$X}_{variable})$$

- Read: There is a fact ("image1.jpg", $X) in photos at myLaptop if there is a fact ("image1.jpg", $X) in photos at picasa

Rules (II)

- Like relations, rules reside at peers
- Rules compute new facts and insert them into relations:

  $$\underbrace{copy@myLaptop(\$X)}_{rulehead} \text{:-} \underbrace{original@myDesktop(\$X)}_{rulebody}$$

- Rules can combine data from multiple relations and peers

  $$friendsBirthdays@myLaptop(\$X, \$Y) \text{:-} friends@facebook(\$X),$$
  $$birthdays@myPhone(\$X, \$Y)$$

- Read: If $X is a friend (according to friends@facebook) and $Y is the birthday of $X (according to birthdays@myPhone) then there is a fact ($X,$Y) in friendsBirthdays@myLaptop. We read the body of a rule left-to-right

Rules (III)

- Given the facts

$$songs@myLaptop(\text{"}Beatles\text{"}, \text{"}Michele\text{"}, \text{"}...\text{"})$$
$$songs@myLaptop(\text{"}Queen\text{"}, \text{"}Flash\text{"}, \text{"}...\text{"})$$
$$songs@yourLaptop(\text{"}Metallica\text{"}, \text{"}One\text{"}, \text{"}...\text{"})$$
$$songs@yourLaptop(\text{"}Nirvana\text{"}, \text{"}Dive\text{"}, \text{"}...\text{"})$$

- A *program* consists of several rules: Copy songs from myLaptop and yourLaptop to hisLaptop

$$songs@hisLaptop(\$X, \$Y, \$Z) \text{:-} songs@myLaptop(\$X, \$Y, \$Z)$$
$$songs@hisLaptop(\$X, \$Y, \$Z) \text{:-} songs@yourLaptop(\$X, \$Y, \$Z)$$

  All songs relations have the schema: $< songs : \text{artist}, \text{title}, \text{content} >$

Rule (IV)

- We can use variables to denote relations and peers
- Given the facts

$$contacts@myLaptop(\text{"}inbox\text{"}, \text{"}annLaptop\text{"}, \text{"}EN\text{"})$$
$$contacts@myLaptop(\text{"}msg\text{"}, \text{"}sueLaptop\text{"}, \text{"}EN\text{"})$$
$$contacts@myLaptop(\text{"}messages\text{"}, \text{"}patLaptop\text{"}, \text{"}FR\text{"})$$

  and relations with the following schemas

$$< contacts : \text{targetRelation}, \text{targetPeer}, \text{language} >$$
$$< inbox : \text{message} >$$
$$< msg : \text{message} >$$
$$< messages : \text{message} >$$

- Send the message: "Hello!" or "Bonjour!" to the contacts

$$\$R@\$P(\text{"}Hello!\text{"}) \text{:-} contacts@myLaptop(\$R, \$P, \text{"}EN\text{"})$$
$$\$R@\$P(\text{"}Bonjour!\text{"}) \text{:-} contacts@myLaptop(\$R, \$P, \text{"}FR\text{"})$$

Examples (I)

1. Copy the music from songs@pandora to songs@iPod
   Answer: $songs@iPod(\$X, \$Y, \$Z)$:-$songs@pandora(\$X, \$Y, \$Z)$
   Schema $< songs : \text{artist}, \text{title}, \text{content} >$

2. Find students who studied CS or Math, given the facts:

   $$roster@college(\text{"}John\text{"}, \text{"}CS\text{"})$$
   $$roster@college(\text{"}John\text{"}, \text{"}Math\text{"})$$
   $$roster@college(\text{"}Ann\text{"}, \text{"}French\text{"})$$
   $$roster@college(\text{"}Sue\text{"}, \text{"}Math\text{"})$$
   $$schemaroster : name, major$$

   Answer:

   $$CSorMath@college(\$X)\text{:-}roster@college(\$X, \text{"}CS\text{"})$$
   $$CSorMath@college(\$X)\text{:-}roster@college(\$X, \text{"}Math\text{"})$$

   Two fact are inserted into CSorMath@college by these rules.

Examples (II)

- Subscribe myLaptop to CNN news
- Answer: at peer CNN

  $$news@\$X(\text{"}cnn\text{"}, \$Y)\text{:-}subscribers@cnn(\$X), news@cnn(\$Y)$$

  add a fact to subscribers@cnn(“myLaptop”)

- Example execution:
  - 9:00am news@cnn(“US Olympic gold”)
  - 9:01am news@myLaptop(“cnn”, ”US Olympic gold”)
  - 9:15am news@cnn(“Higgs boson seen in action”)
  - 9:16am news@myLaptop(“cnn”, ”US Olympic gold”)
  - 9:16am news@myLaptop(“cnn”, “Higgs boson seen in action”)

6.2 Test

We now describe the user study test, that is reproduced literally, except for formatting. For each questions, we also asked the time used to answer.

Problem 1 Consider the following relations and corresponding facts.

```
schema: songs(fileName,content) // the same at all peers

songs@lastFM("song1.mp3", "...")
songs@lastFM("song2.mp3", "...")
songs@lastFM("song3.mp3", "...")
songs@pandora("song4.mp3", "...")
songs@pandora("song5.mp3", "...")
```

Assume that `songs` relations at all peers have the same schema.

1. Write one or several rules that copy all songs from `lastFM` and `Pandora` into relation `songs` at peer `myLaptop`.
2. Suppose now that relation `peers@myLaptop` contains names of peers on which to look for music. You can assume that each peer stores songs in a relation called `songs`, with the same schema as above. Write a WebdamLog program that copies songs from all peers into `songs@myLaptop`.
3. Write a rule that copies songs from `songs@myLaptop` into the `songs` relation on each peer whose name is listed in `peers@myLaptop`.

Problem 2 Consider the following relations and facts.

```
schema: friends(friendName)  photos(fileName,content)
         inPhoto(fileName, friendName)

friends@facebook("ann")
friends@facebook("sue")
friends@facebook("zoe")

photos@ann("sunset.jpg", "...")
photos@ann("vacation.jpg", "...")
photos@ann("party.jpg", "...")

photos@sue("image1.jpg","...")
photos@sue("image2.jpg","...")
```

```
inPhoto@ann("vacation.jpg", "jane")
inPhoto@ann("vacation.jpg", "ann")
inPhoto@ann("party.jpg", "jane")
inPhoto@ann("party.jpg", "zoe")
inPhoto@ann("party.jpg", "sue")

inPhoto@sue("image2.jpg", "sue")
inPhoto@sue("image2.jpg", "jane")
```

Assume that `photos` and `inPhoto` relations at all peers have the same schema. Consider now the following WebdamLog rule.

```
photos@myLaptop($X,$Z) :- friends@facebook($Y),
           photos@$Y($X,$Z), inPhoto@$Y($X,"jane")
```

1. Explain in words what this rule computes.
2. List the facts in that are in `photos@myLaptop` after the rule above is executed.
3. List the facts that are in `photos@myLaptop` if the following rule is executed instead:

```
    photos@myLaptop($X,$Z) :- friends@facebook($Y),
               photos@$Y($X,$Z), inPhoto@$Y($X,"jane"),
               inPhoto@$Y($X,"sue")
```

Problem 3 Recall the example from the tutorial, in which we looked at subscribing the peer `myLaptop` to CNN news. This example is reproduced below.

```
schema: news@cnn(text)    news@myLaptop(source, text)
         subscribers@cnn(peer)
```

```
news@cnn("US Olympic gold")
news@cnn("Higgs boson seen in action")
subscribers@cnn("myLaptop")

[at cnn] news@$X("cnn", $Y) :- subscribers@cnn($X),
                               news@cnn($Y)
```

Suppose that you would now like to receive CNN news on peer `myPhone`, and to store them in relation `news`, with the schema `souce,text`. Describe at least 1 method for doing this. You may assume that you can add rules at peers `cnn`, `myLaptop` and `myPhone`, and that you can insert facts into relations on any of these peers.

6.3 Results

We argued in the introduction that *Webdamlog* can be used to declaratively specify distributed tasks in a variety of applications, including personal data management. The user study to demonstrated the usability of *Webdamlog*.

Participants. We recruited 27 participants for the user study in the US and in France. We present a break-down of results by two groups.

Group 1 consisted of 16 participants with training in Computer Science. Among them, 5 had basic database background, and 4 were familiar with advanced database concepts, including datalog. The group had the following break-down by highest completed education level: 2 highschool, 3 BS, 9 MS, and 2 PhD.

Group 2 consisted of 11 participants with no CS training, and with the following break-down by highest completed education level: 3 vocational school, 6 BS, 2 MS.

Study design. All participants were given a brief tutorial, shown in Section 6.1, in which basic features of *Webdamlog* were explained informally, and demonstrated through examples. On average, getting familiar with the *Webdamlog* language via the tutorial took 15-20 minutes for *Group 1* and 25 minutes for *Group 2*. Following the tutorial, all participants were asked to take a written test, shown in Section 6.2. The three problems were designed to test the comprehension of different features of *Webdamlog*, including local and non-local rules, rules with variable relation and peer names, and delegation.

In the tutorial and the test, we did not make an explicit distinction between intensional and extensional relations, and we ignored recursion.

Results. The results of the study were very encouraging.

Group 1. On Problem 1, 3 participants received a score of 2.5 out of 3, while 13 participants received a perfect score. All participants received a perfect score on Problem 2. Problem 3 was open-ended, and all participants gave at least one correct answer. 4 participants gave 3 correct answers, 4 gave 2 correct answers (2 of these also gave 1 incorrect answer each), and the remaining 8 participants each gave 1 correct answer.

We also asked participants to record how long it took them to answer each problem, in minutes. Problem 1 took between 2.5 and 6 minutes, Problem 2 between 2 and 9 minutes, and Problem 3 between 1 and 8 minutes. We did not observe any correlation between the time it took to answer questions and the participant background in data management or even datalog.

Group 2. On Problem 1, the average score was 2.3, with the following break-down: 6 participants received a perfect score, 3 received 2 out of 3, 1 had a score of 1, and 2 were not able to solve the problem. On Problem 2, 10 participants received a perfect score and 1 got a score of 2 out of 3. On Problem 3, 1 gave 5 good answers, 6 gave 3 good answers, 3 gave 2 good answers, and 2 gave no correct answer. The same two participants failed to answer Problems 1 and 3.

The test took longer for the participants without CS training. Problem 1 took between 6 and 8 minutes to solve in this group, Problem 2 took between 5 and 8 minutes, and Problem 3 took between 4 and 12 minutes.

In summary, all technical and the majority of non-technical participants of our study were able to both understand and write *Webdamlog* programs correctly, with a minimal amount of training. We observed a difference between the technical and non-technical groups in terms of both correctness and time to solution. Two members of the non-technical group were able to understand *Webdamlog* programs but were not able to write programs on their own. We believe that this issue will be alleviated once an appropriate user interface becomes available.

Chapter 7

Conclusion

The philosophy of *Webdamlog* is to return the control of their data to the Web users. When the trend is to entrust more and more data to third-party clouds, *Webdamlog* insists on "Do it yourself", i.e. manage your own data with your own systems. With the concept of delegation, the *Webdamlog* language allows the automation of complex data management tasks, and in particular, those that require the collaboration of several systems. Contrary to proprietary centralized systems, the code of Web is open-source, and *Webdamlog* is based on sharing open code.

Clearly, *Webdamlog* opens a number of directions of research. To conclude this thesis, we mention some that we believe are particularly important:

- In-depth user studies on the usability of *Webdamlog* by regular users (i.e., Web users with little computer science knowledge) would be essential to understand the possibilities and limitations of the approach.
- It would be interesting to develop better interfaces to simplify the task of designing *Webdamlog* applications by regular users.
- Access control for *Webdamlog* programs is a key missing feature towards the full support of personal data management.
- *Webdamlog* encourages the sharing of knowledge between peers or within communities. Clearly such exchanges and integration of data would be facilitated by enhancing *Webdamlog* with ontology technology in the style of semantic Web.
- Finally, we showed how to improve performance using optimization techniques. More is certainly needed to be able to scale to the Web, in terms for instance of number of peers, size of data, and of workload.

Self references

Conferences

[1] Serge Abiteboul, Émilien Antoine, Gerome Miklau, Julia Stoyanovich, and Vera Zaychik Moffitt. Introducing Access Control in Webdamlog. In *DBPL - Proceedings of the 14th International Symposium on Database Programming Languages*, Riva del Garda, Trento, Italie, 2013. `hal.inria.fr/hal-00850754`.

[2] Serge Abiteboul, Émilien Antoine, and Julia Stoyanovich. Viewing the Web as a Distributed Knowledge Base. In *ICDE - Proceedings of the 28th IEEE International Conference on Data Engineering*, Washington DC, United States, 2012. `hal.inria.fr/hal-00703210`.

[3] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Émilien Antoine. A rule-based language for web data management. In *PODS - Proceedings of the 13th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, Athens, Greece, 2011. `hal.inria.fr/inria-00582891`.

Demonstrations

[4] Serge Abiteboul, Émilien Antoine, Gerome Miklau, Julia Stoyanovich, and Jules Testard. Rule-Based Application Development using Webdamlog. In *SIGMOD - Proceedings of the 2013 ACM SIGMOD Special Interest Group on Management Of Data*, New York, United States, 2013. `hal.inria.fr/hal-00817791`.

[5] Serge Abiteboul, Émilien Antoine, Gerome Miklau, Julia Stoyanovich, and Jules Testard. Rule-Based Application Development using Webdamlog. In *BDA - La 29e édition des journées Bases de Données Avancées*, Nantes, France, 2013.

[6] Émilien Antoine, Alban Galland, Kristian Lyngbaek, Amélie Marian, and Neoklis Polyzotis. Social networking on top of the webdamexchange system. In *ICDE - Proceeedings of the 28th IEEE International Conference on Data Engineering*, 2011. `hal.inria.fr/inria-00536361`.

Miscellaneous

[7] Serge Abiteboul, Émilien Antoine, and Julia Stoyanovich. The webdamlog system managing distributed knowledge on the web. Technical report, Inria, 2013. `hal.inria.fr/hal-00813300`.

External references

[AAH+11] Peter Alvaro, Tom J. Ameloot, Joseph M. Hellerstein, William Marczak, and Jan Van den Bussche. A declarative semantics for dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley, Nov 2011.

[AAHM05a] Serge Abiteboul, Zoe Abrams, Stefan Haar, and Tova Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In Chen Li, editor, *Proceedings of the 24th Annual ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'05)*, Baltimore, Maryland, USA, États-Unis, 2005. ACM Press.

[AAHM05b] Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '05, New York, NY, USA, 2005. ACM.

[Aba09] Martín Abadi. Logic in Access Control (Tutorial Notes). In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705, chapter 5. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[ABCM04] Serge Abiteboul, Omar Benjelloun, Bogdan Cautis, and Tova Milo. Active XML, Security and Access Control. In *SBBD*, volume 4, 2004.

[ABGR10] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Marie-Christine Rousset. Distributed datalog revisited. In *Datalog 2.0 Workshop*, 2010.

[Abi03] Serge Abiteboul. Managing an XML warehouse in a P2P context. In *CAiSE*, 2003.

[Abi12] Serge Abiteboul. *Sciences des données: De la logique du premier ordre à la Toile*. Leçons inaugurales du Collège de France. Fayard, 2012.

[ABM04] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive active XML. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '04, New York, NY, USA, 2004. ACM.

[ABM08] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The Active XML project: an overview. *The VLDB Journal*, 17(5), August 2008.

[ABM09] Serge Abiteboul, Pierre Bourhis, and Bogdan Marinoiu. Efficient maintenance techniques for views over active documents. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, New York, NY, USA, 2009. ACM.

[ABMG10] Serge Abiteboul, Pierre Bourhis, Bogdan Marinoiu, and Alban Galland. Axart: enabling collaborative work with axml artifacts. *Proc. VLDB Endow.*, 3, September 2010.

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann Pub, 2000.

[ACC+10] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, New York, NY, USA, 2010. ACM.

[ACG+06] Philippe Adjiman, Philippe Chatalic, Francois Goasdoué, Marie-Christine Rousset, and Laurent Simon. Distributed reasoning in a peer-to-peer setting: Application to the semantic web. *J. Artif. Intell. Res. (JAIR)*, 25, 2006.

[ACHM11] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, 2011.

[Act13] Active-model github, 2013. `https://github.com/rails/rails/tree/master/activemodel/`.

[AD01] Karl Aberer and Zoran Despotovic. Managing trust in a peer-2-peer information system. In *Proceedings of the tenth international conference on Information and knowledge management*, CIKM '01, New York, NY, USA, 2001. ACM.

[ADD+11] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, 5(4), 2011.

[AG94] Miklos Ajtai and Yuri Gurevich. Datalog vs first-order logic. In *Proceedings of the 30th IEEE symposium on Foundations of computer science*, Orlando, FL, USA, 1994. Academic Press, Inc.

[AGM08] Serge Abiteboul, Ohad Greenshpan, and Tova Milo. Modeling the mashup space. In *WIDM '08: Proceeding of the 10th ACM workshop on Web information and data management*, New York, NY, USA, 2008. ACM.

[AGP11] Serge Abiteboul, Alban Galland, and Neoklis Polyzotis. A model for web information management with access control. 14th International Workshop on the Web and Databases, 2011.

[AH08] Dean Allemang and James A. Hendler. *Semantic web for the working ontologist: modeling in RDF, RDFS and OWL*. Morgan Kaufmann, 2008.

[AHV95] Serge Abiteboul, Rick Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AKBC+12] Ahmad Ahmad-Kassem, Christophe Bobineau, Christine Collet, Etienne Dublé, Stéphane Grumbach, Fuda Ma, Lourdes Martínez, and Stéphane Ubéda. Ubiquest, a data-centric approach for networking applications. In *DATA*, 2012.

[AKGU12] Ahmad Ahmad-Kassem, Stéphane Grumbach, and Stéphane Ubéda. Messages with implicit destinations as mobile agents. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, AGERE! '12, New York, NY, USA, 2012. ACM.

[AKSS09] Serge Abiteboul, Benny Kimelfeld, Yehoshua Sagiv, and Pierre Senellart. On the expressiveness of probabilistic XML models. *The VLDB Journal*, 18, October 2009.

[AMC+11] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: datalog in time and space. In *Proceedings of the First international conference on Datalog Reloaded*, Datalog'10, Berlin, Heidelberg, 2011. Springer-Verlag.

[AMP05] Serge Abiteboul, Ioana Manolescu, and Nicoleta Preda. Constructing and Querying Peer-to-Peer Warehouses of XML Resources. In *Semantic Web and Databases*. IEEE, 2005.

[AMP+08] Serge Abiteboul, Ioana Manolescu, Neoklis Polyzotis, Nicoleta Preda, and Chong Sun. XML processing in DHT networks. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, Washington, DC, USA, 2008. IEEE Computer Society.

[AMR+11] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management*. Cambridge University Press, 2011. `http://webdam.inria.fr/textbook`.

[AP07a] Serge Abiteboul and Neoklis Polyzotis. The data ring: Community content sharing. In *CIDR*, 2007.

[AP07b] Serge Abiteboul and Neoklis Polyzotis. The data ring: Community content sharing. In *Conference on Innovative Data Systems Research (CIDR)*, 2007.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1), April 1997.

[ASV09] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of active XML systems. *ACM Trans. Database Syst.*, 34, December 2009.

[AV91] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1), August 1991.

[AvH08] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer, 2nd Edition (Cooperative Information Systems)*. The MIT Press, 2 edition, 2008.

[AVM07] Bader Ali, Wilfred Villegas, and Muthucumaru Maheswaran. A trust based approach for protecting user data in social networks. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, New York, NY, USA, 2007. ACM.

[BAP+12] Harold Boley, Tara Athan, Adrian Paschke, Said Tabet, Benjamin Grosof, Nick Bassiliades, Guido Governatori, Frank Olken, and David Hirtle. Schema specification of deliberation ruleml. ruleml.org/spec/, April 2012.

[BCGR98] Elisa Bertino, Barbara Catania, Vincenzo Gervasi, and Alessandra Raffaetà. Active-u-datalog: Integrating active rules in a logical update language. In Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998.

[BFG07] Moritz Becker, Cedric Fournet, and Andrew Gordon. Design and Semantics of a Decentralized Authorization Language. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, Washington, DC, USA, 2007. IEEE Computer Society.

[Bir05] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2002. ACM.

[BLC90] Tim Berners-Lee and Robert Cailliau. WorldWideWeb: Proposal for a hypertexts project. `http://www.w3.org/Proposal.html`, November 1990.

[BM10] Dan Brickley and Libby Miller. Foaf vocabulary specification 0.98. `http://xmlns.com/foaf/spec/`, August 2010.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '86, New York, NY, USA, 1986. ACM.

[Bry05] Jery Bryans. Reasoning about XACML policies using CSP. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, New York, NY, USA, 2005. ACM.

[BSVD09] Sonja Buchegger, Doris Schiöberg, Le H. Vu, and Anwitaman Datta. PeerSoN: P2P social networking: early experiences and insights. In *SNS '09: Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, New York, NY, USA, 2009. ACM.

[BT07] Peter Buneman and Wang C. Tan. Provenance in databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2007. ACM.

[Byn13] Mathias Bynens. Javascript, aka. web ecmascript standard. `http://javascript.spec.whatwg.org/`, June 2013.

[CCHM08] Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. Evita raced: metacompilation for declarative networks. *Proc. VLDB Endow.*, 1(1), 2008.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[CGL09] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, New York, NY, USA, 2009. ACM.

[CH85] Ashok K. Chandra and David Harel. Horn clause queries and generalizations. *The Journal of Logic Programming*, 2(1), April 1985.

[CKW93] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. *JOURNAL OF LOGIC PROGRAMMING*, 15(3), 1993.

[CR93] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, New York, NY, USA, 1993. ACM.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, chapter 4. Springer Berlin / Heidelberg, Berlin, Heidelberg, March 2001.

[Dat10] Datalog 2.0 workshop. `http://www.datalog20.org/`, 2010. Oxford Univ.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, volume 41 of *SOSP '07*, New York, NY, USA, 2007. ACM.

[Dia] Diaspora. `https://diasporafoundation.org/`.

[EK76] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23, 1976.

[ERC13] ERC grant Webdam, 2009-2013. `webdam.inria.fr`.

[Eve13] Event-machine github, 2013. `https://github.com/eventmachine/eventmachine/`.

[FHM05] Michael J. Franklin, Alon Y. Halevy, and David Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4), 2005.

[FHMV03] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about knowledge*. The MIT Press, 2003.

[FJ02] Csilla Farkas and Sushil Jajodia. The inference problem: a survey. *SIGKDD Explor. Newsl.*, 4, December 2002.

[FMS09] John Field, Maria C. Marinescu, and Christian Stefansen. Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. *Theor. Comput. Sci.*, 410, February 2009.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Gal11] Alban Galland. *Distributed data management with access control : social Networks and Data of the Web*. These, Université Paris Sud - Paris XI, September 2011.

[GKIT07] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07. VLDB Endowment, 2007.

[GKIT10] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Provenance in orchestra. *IEEE Data Eng. Bull.*, 33(3), 2010.

[GKT07] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, New York, NY, USA, 2007. ACM.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, 1988.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), June 2002.

[GW10] Stéphane Grumbach and Fang Wang. Netlog, a rule-based language for distributed programming. In *PADL*, 2010.

[Hel10] Joseph M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39(1), 2010.

[HNN09] Richard Hull, Nanjangud Narendra, and Anil Nigam. Facilitating Workflow Interoperation Using Artifact-Centric Hubs. In Luciano Baresi, Chi-Hung Chi, and Jun Suzuki, editors, *Service-Oriented Computing*, volume 5900, chapter 1. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Hul89] G. Hulin. Parallel processing of recursive queries in distributed architectures. In *Proceedings of the 15th international conference on Very large data bases*, VLDB '89, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[HZ96] Richard Hull and Gang Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1996. ACM.

[Int12] Matteo Interlandi. Knowlog: A declarative language for reasoning about knowledge in distributed systems. In *ER*, 2012.

[ISO99] ISO. Sql 3 specification. `http://www.iso.org/`, 1999.

[JOV05] H. V. Jagadish, Beng C. Ooi, and Quang H. Vu. BATON: a balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05. VLDB Endowment, 2005.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, volume 28, New York, NY, USA, December 2000. ACM.

[KGG+06] Sebastian Kruk, Sławomir Grzonkowski, Adam Gzella, Tomasz Woroniecki, and Hee-Chul Choi. D-FOAF: Distributed Identity Management with Access Rights Delegation. In Riichiro Mizoguchi, Zhongzhi Shi, and Fausto Giunchiglia, editors, *The Semantic Web – ASWC 2006*, volume 4185 of *Lecture Notes in Computer Science*, chapter 15. Springer Berlin Heidelberg, 2006.

[Kif08] Michael Kifer. Rule interchange format: The framework. In Diego Calvanese and Georg Lausen, editors, *Web Reasoning and Rule Systems*, volume 5341 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008.

[KIT10] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD*

International Conference on Management of data, SIGMOD '10, New York, NY, USA, 2010. ACM.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, New York, NY, USA, 1997. ACM.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4), July 1995.

[Kol05] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, New York, NY, USA, 2005. ACM.

[KW94] Brigitte Kröll and Peter Widmayer. Distributing a search tree among a growing number of processors. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, SIGMOD '94, New York, NY, USA, 1994. ACM.

[LCG+06] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[LCG+09] Boon T. Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Commun. ACM*, 52(11), November 2009.

[LCH+05] Boon T. Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5), October 2005.

[LFWK09] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. Openrulebench: an analysis of the performance of rule engines. In *WWW*, 2009.

[LHSR05] Boon T. Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with

declarative queries. *SIGCOMM Comput. Commun. Rev.*, 35, August 2005.

[LIJ+13] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. Dbpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 2013. Under review.

[Lit80] Witold Litwin. Linear hashing: a new tool for file and table addressing. In *Proceedings of the sixth international conference on Very Large Data Bases - Volume 6*. VLDB Endowment, 1980.

[LLM98] Georg Lausen, Bertram Ludäscher, and Wolfgang May. On Active Deductive Databases: The Statelog Approach. In Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *Lecture Notes in Computer Science*. Birkhäuser Basel, 1998.

[LM75] K. Dan Levin and Howard L. Morgan. Optimizing distributed data bases: a framework for research. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, AFIPS '75, New York, NY, USA, 1975. ACM.

[LMO+08] Changbin Liu, Yun Mao, Mihai Oprea, Prithwish Basu, and Boon T. Loo. A declarative perspective on adaptive manet routing. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '08, New York, NY, USA, 2008. ACM.

[LNS94] Witold Litwin, Marie A. Neimat, and Donovan A. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[LNS96] Witold Litwin, Marie A. Neimat, and Donovan A. Schneider. LH* a scalable, distributed data structure. *ACM Trans. Database Syst.*, 21(4), December 1996.

[LTZ+09] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. In *ICDE*, 2009.

[Lud98] Bertram Ludäscher. *Integration of Active and Deductive Database Rules*, volume 45 of *DISDBIS*. Infix Verlag, St. Augustin, Germany, 1998.

[MAC+12] William R. Marczak, Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Confluence analysis for distributed programs: a model-theoretic approach. In *Proceedings of the Second international conference on Datalog in Academia and Industry*, Datalog 2.0'12, Berlin, Heidelberg, 2012. Springer-Verlag.

[MFF+08] Luc Moreau, Juliana Freire, Joe Futrelle, Robert McGrath, Jim Myers, and Patrick Paulson. The open provenance model: An overview. In Juliana Freire, David Koop, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 5272 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-89965-5_31.

[MHB+10] William R. Marczak, Shan Shan Huang, Martin Bravenboer, Micah Sherr, Boon Thau Loo, and Molham Aref. Secureblox: customizable secure distributed data processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, New York, NY, USA, 2010. ACM.

[MMSW07] Maged Michael, Jose E. Moreira, Doron Shiloach, and Robert W. Wisniewski. Scale-up x Scale-out: A Case Study using Nutch/Lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[MS02] Gerome Miklau and Dan Suciu. Cryptographically Enforced Conditional Access for XML. In *Fifth International Workshop on the Web and Databases (WebDB*, 2002.

[MS03] Gerome Miklau and Dan Suciu. Controlling access to published data using cryptography. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*. VLDB Endowment, 2003.

[MSM+12] Atsuyuki Morishima, Norihide Shinagawa, Tomomi Mitsuishi, Hideto Aoki, and Shun Fukusumi. Cylog/crowd4u: a declarative platform for complex data-centric crowdsourcing. *Proc. VLDB Endow.*, 5(12), August 2012.

[MZZ+08] William R. Marczak, David Zook, Wenchao Zhou, Molham Aref, and Boon T. Loo. Declarative Reconfigurable Trust Management. In *Conference on Innovative Data Systems Research (CIDR)*, 2008.

[NC03] Anil Nigam and Nathan S. Caswell. Business artifacts: An approach to operational specfication. In *IBM Systems Journal, vol. 42, no. 3*, 2003.

[NCR08] G. H. Nguyen, P. Chatalic, and M. C. Rousset. A probabilistic trust model for semantic peer to peer systems. In *DaMaP '08: Proceedings of the 2008 international workshop on Data management in peer-to-peer systems*, New York, NY, USA, 2008. ACM.

[NCW93] Wolfgang Nejdl, Stefano Ceri, and Gio Wiederhold. Evaluating recursive queries in distributed databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(1), February 1993.

[NR09] Juan Navarro and Andrey Rybalchenko. Operational Semantics for Declarative Networking. In Andy Gill and Terrance Swift, editors, *Practical Aspects of Declarative Languages*, volume 5418 of *Lecture Notes in Computer Science*, chapter 6. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009.

[OAS04] OASIS. Uddi version 3.0.2. `http://uddi.org/pubs/uddi_v3.htm`, October 2004.

[OAS07] OASIS. Web services business process execution language version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, April 2007.

[oI] University of Innsbruck. IRIS – integrated rule inference system. `http://iris-reasoner.org/`.

[oUB] Bloom Team of UC Berkeley. BFS – Bloom Filesystem. `https://github.com/bloom-lang/bud-sandbox/tree/master/bfs`.

[ÖV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1999.

[PRS09] Juan A. Pérez, Andrey Rybalchenko, and Atul Singh. Cardinality Abstraction for Declarative Networking Applications. In *CAV '09: Proceedings of the 21st International Conference on*

Computer Aided Verification, Berlin, Heidelberg, 2009. Springer-Verlag.

[Prz90] Teodor C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundam. Inform.*, 13(4), 1990.

[Rai13] Rails github, 2013. `https://github.com/rails/rails/`.

[RFC74] RFC675. Specification of internet transmission control program. `http://tools.ietf.org/html/rfc675`, December 1974.

[RS09] Royi Ronen and Oded Shmueli. Evaluating very large datalog queries on social networks. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, New York, NY, USA, 2009. ACM.

[Rub] Ruby community. ERB documentation standard. `http://ruby-doc.org/stdlib-2.0.0/libdoc/erb/rdoc/ERB.html`.

[Sch95] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching, 1995.

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, New York, NY, USA, 2007. ACM.

[SW85] Domenico Sacca and Gio Wiederhold. Database partitioning in a cluster of processors. *ACM Trans. Database Syst.*, 10(1), March 1985.

[TS04] Stephanos A. Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4), December 2004.

[Vie86] Laurent Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proc. 1st Int. Conf. on Expert Database Systems*, 1986.

[W3C99] W3C. Xsl transformations (xslt) standard version 1.0. `http://www.w3.org/TR/xslt`, November 1999.

[W3C02] W3C. Web services conversation language (wscl) 1.0 standard. `http://www.w3.org/TR/wscl10/`, March 2002.

[W3C04a] W3C. Rdf primer standard. http://www.w3.org/TR/rdf-primer/, February 2004.

[W3C04b] W3C. Xml schema part 0: Primer standard. http://www.w3.org/TR/xmlschema-0/, October 2004.

[W3C07a] W3C. Soap version 1.2 part 1: Messaging framework (second edition) standard. http://www.w3.org/TR/soap12-part1/, April 2007.

[W3C07b] W3C. Web services description language (wsdl) standard version 2.0 part 1: Core language. http://www.w3.org/TR/wsdl20/, June 2007.

[W3C08a] W3C. Extensible markup language (xml) 1.0 standard. http://www.w3.org/TR/REC-xml/, November 2008.

[W3C08b] W3C. Xml signature syntax and processing (second edition) standard. http://www.w3.org/TR/xmldsig-core/, June 2008.

[W3C09] W3C. Owl 2 web ontology language document overview. http://www.w3.org/TR/owl2-overview/, October 2009.

[W3C10] W3C. Xquery 1.0: An xml query language (second edition) standard. http://www.w3.org/TR/xquery/, December 2010.

[W3C13] W3C. Html 5.1 specification. http://www.w3.org/TR/html51/, May 2013.

[Wal03] Dan Wallach. A Survey of Peer-to-Peer Security Issues. In Mitsuhiro Okada, Benjamin Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, *Software Security — Theories and Systems*, volume 2609 of *Lecture Notes in Computer Science*, chapter 4. Springer Berlin / Heidelberg, Berlin, Heidelberg, June 2003.

[WL82] Paul F. Wilms and Bruce G. Lindsay. A database authorization mechanism supporting individual and group authorization. In *Distributed data sharing systems: proceedings of the Second International Seminar on Distributed Data Sharing Systems*, June 1982.

[YHY07] Xiaoxin Yin, Jiawei Han, and Philip S. Yu. Truth discovery with multiple conflicting information providers on the web. In

KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, New York, NY, USA, 2007. ACM.

[YK00] Guizhen Yang and Michael Kifer. Flora: Implementing an efficient dood system using a tabling logic engine. In *In International Conference on Computational Logic, volume 1861 of LNCS*, 2000.

[YKZ03] Guizhen Yang, Michael Kifer, and Chang Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *In Second International Conference on Ontologies, Databases and Applications of Semantics (ODBASE*, 2003.

[ZFS+11] Wenchao Zhou, Qiong Fei, Shengzhi Sun, Tao Tao, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. Nettrails: a declarative platform for maintaining and querying provenance in distributed systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, New York, NY, USA, 2011. ACM.

[ZST+10] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, New York, NY, USA, 2010. ACM.