Query rewriting using views : a theoretical and practical perspective

Ioana Ileana

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2014-ENST-0062

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Ioana ILEANA

le 24 Octobre 2014

Réécriture de requêtes avec des vues : une perspective théorique et pratique

Directeur de thèse : **Bogdan CAUTIS**
Co-encadrement de la thèse : **Pierre SENELLART**

Jury

Mme Angela BONIFATI, Professeur, LIFL, Université Lille 1 et Inria Lille		Rapporteur
M. Dan OLTEANU, Associate Professor, Dpt. of Computer Science, Oxford University		Rapporteur
Mme Ioana MANOLESCU, Directeur de recherches, LRI, Inria Saclay et Univ. Paris-Sud		Examinateur
M. Alin DEUTSCH, Professeur, Dpt. of Computer Science and Electrical Eng., UCSD		Examinateur
M. Bernd AMANN, Professeur, LIP6, Université Pierre et Marie Curie		Examinateur
M. Fabian SUCHANEK, Maître de conférences, DBWeb, Télécom ParisTech		Examinateur
M. Bogdan CAUTIS, Professeur, LRI, Univ. Paris-Sud et Inria Saclay		Examinateur
M. Pierre SENELLART, Professeur, DBWeb, Télécom ParisTech		Examinateur

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

THÈSE

To Damien, Dad, Mum and Lidia
(certainly the most patient people in the world)

Acknowledgements

I would like to thank my reviewers and jury for accepting to be my reviewers and jury, and thus committing to all the work and travel (and headache) this status implies. I also thank my PhD advisors and the DBWeb team in Telecom ParisTech.

This manuscript has undergone an amount of changes, and I owe a tremendous quantity of improvements and refinements to Angela and Dan's reviews. Beyond the formal review system, such feedback allowed me to importantly improve the structure and clarity of my writing, so I would like to thank them for the huge formative impact of their feedbacks.

I achieved a very significant part of my knowledge during these three PhD years thanks to Alin Deutsch's advising and mentoring. Working with Alin has provided me the chance to dwelve into some of the most interesting topics of my PhD, which I intend to further pursue in future research work. I would also like to thank Alin, as well as Pierre, Ioana, Fabian and Nicoleta, for their extensive help and advising regarding my career choices and approach of academia.

Last but certainly not least, I owe the finish line of this thesis, as all my achievements, to the constant and faultless support of my family (to whom I dedicate this manuscript) and my friends.

Abstract

The massive amounts of data available nowadays generate an increasing need for optimizing and speeding up data search and access. Among the most efficient known accelerators, materialized views have proven for a long time their benefit in speeding up queries, quite often dramatically. Using views for decreasing data search and access cost raises however a range of complex problems, amongst which the question of whether and how existing materialized views can be used to answer a given query: in other words, the problem of *query rewriting using views*.

This problem can be further placed in the general framework of *query reformulation*: given a query Q expressed against a source schema S, find an equivalent query R formulated against a target schema T, by exploiting the relationship between S and T. Accordingly, views can be seen not only as data access accelerators, but more generally as data access *models*. This is for instance the case of scenarios such as security-restricted access through views, data pricing, mediator-like, multi-storage and multi-model architectures.

In this work, we address the problem of query rewriting using views, by adopting both a theoretical and a pragmatic perspective. In the first and main chapter, we approach the topic of finding all minimal (i.e. with no redundant relational atoms) conjunctive query reformulations for a relational conjunctive query, under constraints expressed as embedded dependencies, including the relationship between the source and the target schemas. We present a novel sound and complete algorithm, the Provenance-Aware Chase & Backchase, that solves the minimal reformulations problem with practically relevant performance. We provide a detailed theoretical characterization of our algorithm. We further present the optimized implementation and the experimental evaluation thereof, and exhibit natural scenarios yielding speed-ups of up to two orders of magnitude between the execution of a best view-based rewriting found by a commercial DBMS and that of a best rewriting found by our algorithm. We generalize the Provenance-Aware Chase & Backchase towards directly finding minimum-cost reformulations for monotonic cost functions, and show the performance improvements this adaptation further enables. With our algorithm, we introduce a novel chase flavour, the Provenance-Aware Chase, which is interesting on its own, as a means of reasoning about the interaction between provenance and constraints.

In the second chapter, we move to an XML context and revisit the previous work of Cautis, Deutsch and Onose on the problem of finding XPath query rewritings with a single level of intersection of multiple views. We enrich the analysis of the rewriting problem by showing its links to the problems of DAG-tree equivalence and union-freeness. We refine the rule-based rewriting technique proposed by Cautis, Deutsch and Onose to ensure its polynomial complexity and improve its completeness, and present a range of optimizations on the rewriting procedures, necessary to achieve practical performance. We provide a complete implementation comprising these optimizations and a thorough experimental evaluation thereof, showing the performance and utility of the polynomial rewriting technique.

Résumé

La quantité massive de données disponibles de nos jours génère un besoin croissant d'optimisation et accélération de la recherche et de l'accès à ces données. Les vues matérialisées s'imposent depuis longtemps en tant qu'accélérateur souvent très important des requêtes. Leur utilisation engendre toutefois une gamme de problèmes complexes, parmi lesquels la question de savoir si et comment des vues existantes peuvent être utiles pour répondre à une requête - autrement dit, le problème de la *réécriture de requêtes avec des vues*.

Ce problème peut aussi être placé dans le cadre général de la *reformulation de requêtes* : étant donnée une requête Q exprimée par rapport à un schéma source S, trouver une requête équivalente R formulée sur un schéma cible T, en exploitant la relation entre S et T. Dans ce contexte, les vues peuvent être considérées non seulement comme des accélérateurs, mais de façon plus générale comme des *modèles* de l'accès aux données. Ceci est le cas par exemple dans des scénarios de restrictions de sécurité de l'accès par des vues ou de prix associé aux données, ou dans les architectures multi-stockage et multi-modèles de données.

Dans ce document, nous adressons le problème de la réécriture de requêtes avec des vues, en adoptant une perspective à la fois théorique et pratique. Dans le premier et principal chapitre, nous approchons le sujet de la recherche de toutes les reformulations minimales (sans atomes relationnels redondants) pour une requête relationnelle conjonctive, sous des contraintes d'intégrité qui incluent la relation entre les schémas source et cible. Nous présentons un nouvel algorithme, correct et complet, le Provenance-Aware Chase & Backchase, qui résout le problème des reformulations avec des performances significatives sur le plan pratique. Nous présentons sa charactérisation théorique detaillée, son implémentation optimisée et son évaluation, montrant des gains de performance jusqu'à deux ordres de grandeur par rapport à un SGBD commercial. Nous généralisons notre algorithme pour trouver directement des reformulations de coût minimum pour les fonctions de coût monotones, et montrons les gains de performance de cette adaptation. Avec notre algorithme, nous introduisons également un nouveau type de *chase*, la Provenance-Aware Chase, qui comporte son propre intérêt théorique, en tant que moyen de raisonnement sur l'interaction entre la provenance et les contraintes.

Dans le deuxième chapitre, nous nous plaçons dans un contexte XML et nous revisitons le travail de Cautis, Deutsch and Onose sur problème de la réécriture de requêtes XPath par un seul niveau d'intersection de plusieurs vues. Nous étendons l'analyse de ce probleme en montrant ses connexions avec les problèmes de l'équivalence DAG-arbre et de la *union-freeness* d'un DAG. Nous raffinons un algorithme de réécriture proposé par Cautis, Deutsch and Onose pour obtenir une complexité polynomiale et améliorer sa complétude, et présentons un ensemble d'optimisations des procedures de réécriture, necessaires pour atteindre des performances pratiques. Nous fournissons une implementation complète comprenant ces optimizations ainsi que son evaluation éxperimentale extensive, montrant la performance et l'utilité de la technique polynomiale de réécriture.

Contents

Introduction

Data search and access: a pragmatic perspective

We are nowadays surrounded by data: personal and enterprise data, sensor-collected data, massive amounts of data coming from the Web and social networks. *Big data* has become a commonplace in mainstream vocabulary. With the ever increasing storage and computational technologies available, big data seems easier and easier to digest, and there is a rising tendency of taking for granted the resources involved in searching and accessing any piece of data.

The cost is however there, and indeed data search and access comes at a cost, even when one owns the data. From a plain financial angle, this cost can be seen as the prohibitive price of storage and processing equipment, or alternatively access fees for various cloud-based services, when the storage and processing power is outsourced. As soon as resources are limited for financial purposes, the cost becomes visible as the *slowness* in the search and access of data: suddenly, one is facing a long wait for the execution of a medium-sized SQL query over a medium-sized database. This may come as a surprise for the Google user accustomed to an instantaneous answer to his/her query over the vast World Wide Web.

It is typically in these settings that the need for optimization in data search and access finds its way back to the spotlight. One realizes the need for clever algorithms that allow diminishing storage and transfer and speeding up search, without involving additional resources. Search and access accelerators like caches, materialized views and indexes come back into focus, after being shadowed by the fake certitude that querying and accessing (big) data is inherently fast. Any opportunity for optimization in practice becomes a desired goal: efficient in-memory processing, polynomial or even less exponential algorithms with efficient and well-adapted implementations.

Materialized views as a data search and access enhancer

Among the search and access accelerators, materialized views and caches have been for a long time known for their benefit in speeding up queries, quite often dramatically. While the term *views* has a database-related connotation, *cache* is an ubiquitous term nowadays, omnipresent in topics related to Web servers and clients. Both concepts express the notion of short-circuiting some costly remote access and/or some costly processing involved in the search of data, by (locally) materializing pre-computed results.

Using views for decreasing the cost of data search and access raises however a range of questions, such as which views should be materialized for *best access efficiency*, and how should these views be efficiently maintained up-to-date. Furthermore, to achieve a gain in performance

by relying on materialized views, the cost of selecting and maintaining them should be such that it is largely counterbalanced by the speed-up obtained by employing these views for searching and accessing data. Even when all these issues are dealt with, there remains the paramount question of whether and how existing materialized views can be used to answer a given query - in other words, the problem of *query rewriting using views*.

Rewriting using views and query reformulation

Besides the above classic query optimization scenarios, where the purpose is to accelerate query execution by relying on previously materialized views, view-based rewriting can be further placed in the general framework of *query reformulation*: given a query Q expressed against a source schema S, find an equivalent query R formulated against a target schema T, by exploiting the relationship between S and T. Query reformulation further includes several other problems that have occupied database research and practice for decades, such as physical access path selection and semantic optimization (e.g. redundant join elimination and other instances of rewriting queries under integrity constraints).

Accordingly, views can be seen not only as data access accelerators, but more generally as data access models. For instance, beyond the caching properties, views can also express secured entry points in a context of security restrictions. In this case, the access through views is not aimed at because of potential speed-up, but because it becomes *the only possible access*. A similar, though more refined setting involves *data pricing* scenarios, when the access is not only restricted but furthermore priced differently according to the views employed. Moreover, mediator-like and multi-storage or multi-model architectures can also be modelled using views and thus provide a variety of practical settings of query reformulation and view-based rewritings.

Outline and contributions

In this work, we address the problem of query rewriting using views, by adopting both a theoretical and a pragmatic point of view. We place an important focus on theoretical analysis, correctness and complexity; in the same time, we are constantly driven by a pragmatic perspective, and many of our theoretical developments stem from the need of achieving *practical performance*.

In the **first and main chapter** of this thesis, we approach the topic of **finding minimal conjunctive query reformulations for relational conjunctive queries, under integrity constraints**, where these constraints include (but are not limited to) the relationship between the source and the target schema. A minimal reformulation is such that it does not contain in the FROM clause elements that are redundant, unnecessary for ensuring equivalence with the to-be-reformulated query, under the given constraints.

All the reformulation algorithms we are interested in throughout this work are expected to be sound, that is, to return correct reformulations (equivalent to the input query). In our approach of the minimal reformulations problem, we further place a major focus on achieving *completeness*. In general, for a reformulation algorithm, its completeness (or *strong completeness*) with respect

to a class of solutions means the capacity of finding *all reformulations* in the given class. The immediate and central interest of finding all minimal reformulations is that, under reasonable cost models, the minimum-cost reformulations will always be a subset of the minimal ones.

Completeness is thus clearly desirable for practical scenarios that define a certain measure of a query as the overall minimum across *all* its reformulations. For instance, consider the case of access control enforcement via *security views* [54, 61], where a query is considered safe only if it has a total rewriting using a set of safe views. In previous work, the existence of such rewriting sufficed for the query to be allowed to run (against the base tables, the safe views being virtual). Let's refine the scenario by having each view require a certain clearance level, and assume that an analyst wishes to establish the minimum clearance level required to answer a query so he can go request it. This involves then finding all possible total rewritings and selecting the minimum-clearance one(s) among them. The same reasoning can be developed for *data pricing* [43] scenarios, in which the data owner sets the price for several views over his data. Subsequent queries can then be priced automatically whenever they are determined by the priced views, such that the query price is that of the cheapest total rewriting. Completeness is also essential in classical optimization, as the best reformulation among those inspected by an incomplete algorithm can be significantly worse than the optimum one(s), which a complete reformulation algorithm is guaranteed to find. Indeed, as our experiments show, even the best reformulation found by a sophisticated commercial relational optimizer in a natural setting involving materialized views can execute orders of magnitude slower than an optimum reformulation.

However, given that the particular case of reformulation corresponding to total view-based rewritings of a query has an NP-hard associated decision problem even in the absence of constraints [46], conventional wisdom has held so far that completeness is likely to remain a concept of mainly theoretical interest. Indeed, for the Chase & Backchase [27], which is the only complete algorithm we are aware of in this context, the search for minimal reformulations does not scale beyond the low end of the spectrum of practically occurring query and constraint set sizes. The reason is that, even when there are few actual minimal reformulations for a query, the $C\&B$ inspects a number of candidate reformulations that is often exponential in the size of the query and number of views, thus launching exponentially many chases. [60] confirms this fact experimentally, then dedicates the bulk of the results to heuristics that dramatically reduce the search space for minimal reformulations by trading completeness for search speed. Similar trade-offs are adopted by all other existing implementations for query reformulation, including the optimizers of relational DBMSs and the follow-up $C\&B$-based implementations for XML query reformulation in [29, 57, 70].

In this work, we challenge conventional wisdom and hardness results by presenting a **novel sound and complete algorithm**, the Provenance-Aware Chase & Backchase, that solves the minimal reformulations problem with **practically relevant performance**. We provide its **detailed theoretical characterization and its optimized implementation**. We further present its **experimental evaluation**, and exhibit natural scenarios yielding speed-ups of up to two orders of magnitude between the execution of a best view-based rewriting found by a commercial DBMS and that of a best rewriting found by $Prov_{C\&B}$ (which the DBMS misses because of *incomplete* reasoning about reformulations). We further show how to **adapt our algorithm to-**

wards directly finding minimum-cost reformulations for monotonic cost functions, and the performance gains this adaptation can further induce.

The Provenance-Aware Chase & Backchase transforms the standard Chase & Backchase algorithm by employing a much more directed, goal-oriented technique for the search of reformulations. The main reason for the performance achieved by $Prov_{C\&B}$ is the fact that the potentially exponential number of chases in the original Chase & Backchase is replaced in the $Prov_{C\&B}$ by a single chase, employing **a novel chase technique, the Provenance-Aware Chase**. As its name implies, the Provenance-Aware Chase is a chase procedure that employs provenance instrumentation, such that the provenance annotations it produces and maintains reflect the minimal reformulations we are interested in. The particular provenance flavour employed corresponds to the minimal-why provenance, introduced for a different purpose in [14]. The design of the Provenance-Aware Chase has been technically challenging, as the standard chase is not directly suited for provenance instrumentation, creating the need for the design of an additional, provenance-agnostic chase flavour, which we call the Conservative Chase. In its statement as the Conservative Chase with provenance annotations, besides its usage in $Prov_{C\&B}$, the Provenance-Aware Chase becomes interesting on its own, as a means of **reasoning about the interaction between provenance and constraints**.

In the **second chapter** of this thesis we move to an XML context and revisit the previous work by Cautis, Deutsch and Onose, presented in [16] and detailed in [56], on **XPath rewritings using a single level of intersection of multiple views**: that is, rewritings where one first navigates inside the views, then intersection occurs, and then potential additional navigation may be applied. The work we revisit provides a complexity analysis for the rewriting problem in this setting, as well as a sound and complete algorithm for its resolution. Compared to the setting analysed in the first chapter, the completeness concept targeted is one of *weak completeness*: an algorithm is complete in this case if it finds *at least one reformulation* in a given class C whenever one exists. Note that a weakly complete reformulation algorithm can serve as a decision procedure for the problem of existence of a reformulation from C, but goes beyond the requirements of the decision problem by outputting the reformulation. In the case of the XPath rewriting setting we revisit, this behaviour is desirable and useful for instance for scenarios of security-restricted access through views, as those mentioned above (without any cost refinement), where the access through views is the only possible access, and it is essential to find such an access as soon as one is available.

Our main motivation for the work presented in the second chapter is that of investigating and achieving practical performance. Following the proven hardness results, [16] presents and [56] details the usage of a rule-based procedure for inferring an additional sound algorithm that solves the rewrite problem, as well as conditions for this sound algorithm to become complete. We **refine** this rule-based procedure to ensure its **polynomial complexity** and **enhance the completeness conditions** of the corresponding rewriting algorithm. We further present a **range of optimisations** of the rewriting techniques, necessary in order to achieve **practical performance**. We provide **a complete implementation** of the rewriting techniques comprising these refinements and optimisations, and further present a thorough **experimental evaluation** thereof,

which shows the practical performance and benefits of the refined and optimized polynomial rewriting procedure.

As a side effect of our review of the work in [16] and [56], we also **enrich the analysis of the rewriting problem** by showing, structuring and clarifying its connections to the problem of deciding the equivalence between a query expressed by a DAG pattern and a query expressed by a tree pattern, and further to the problem of union-freeness, i.e. finding any tree pattern query equivalent to a DAG pattern query.

The first chapter of this thesis builds on, refines and extends our paper [42]: **Ileana, Cautis, Deutsch, Katsis,** ***Complete yet practical search for minimal query reformulations under constraints***, SIGMOD Conference 2014, 1015-1026.

Our cost-based refined version of $Prov_{C\&B}$ as described in the first chapter is further intended to provide a main brick of the ESTOCADA system, presented in the paper (currently under review for CIDR 2015): **Bugiotti, Bursztyn, Deutsch, Ileana, Manolescu,** ***Invisible Glue: Scalable Self-Tuning Multi-Stores***.

Finally, the second chapter refines and extends our contribution to the journal paper (currently under review for TCS): **Cautis, Deutsch, Ileana, Onose:** ***Rewriting XPath queries using view intersections: tractability versus completeness***.

Other topics explored during this PhD

While this manuscript's main focus is on view-based rewritings, this PhD comprises additional work on several other topics belonging to the broader range of query accelerators.

The main two such topics, explored in detail and presented in Appendix A, are related to *indexing*. The first one, provided by the **ACM SIGMOD Programming Contest 2012**, involves the construction of a **multidimensional**, high-throughput, in-memory **index** structure, supporting common database operations such as point and range queries as well as data manipulation, in a **highly concurrent** setting consisting of many client threads operating queries and updates in parallel. We present in Section A.1 our work on this topic, which has been rewarded with the **second prize in the contest**.

The second indexing-related topic concerns **structured Web sources indexing and selection** for Web wrapper inference. Structured Web sources are sets of web pages exhibiting similar, structured contents, such as the Amazon book pages. Web wrapping involves the extraction of the pages' relevant data by relying on their common structure. Web source selection supposes a user-provided *lightweight description of the type of data that is targeted* and its usage for selecting, via an index-based structure, a subset of previously crawled sources matching this data requirement. We present in Section A.2 our work on this topic, developing previous work by Derouiche, Cautis and Abdelssalem, and supported by the Arcomem project.

Finally, a third topic explored is one that spans both indexing strategies and view-based rewriting, and concerns the problem of **view indexing** to the purpose of speeding-up the rewriting computation. While currently in an early development stage, we believe this topic to be of interest and worth pursuing in future work, as a means of providing complementary performance enhancements to our reformulation approaches presented in this manuscript.

Chapter 1

A complete yet practical algorithm for finding minimal query reformulations under constraints

We present in this chapter the *Provenance-Aware Chase & Backchase* algorithm ($Prov_{C\&B}$) for finding minimal conjunctive query reformulations for conjunctive queries, under constraints. The $Prov_{C\&B}$ revisits the classic Chase & Backchase ($C\&B$) algorithm [27] with a clear and simple aim: preserve *completeness* (a paramount feature of the original $C\&B$) but *at practically relevant performance* (which the original $C\&B$ fails to achieve).

We recall the problem of query reformulation, as presented in the introductory section: given a query Q, formulated against a source schema S, find an equivalent query R formulated against a target schema T, by exploiting the relationship between S and T. The authors of the $C\&B$ start from the observation that in an important range of instances of the query reformulation problem, the relationship between the source and the target schemas can be expressed by *constraints*. They then present a uniform and generalized solution to such problems, in the form of the $C\&B$ algorithm, which finds all the minimal reformulations under a set of constraints that includes, but is not limited to, the relationship between the schemas S and T. The $C\&B$ applies to relational conjunctive queries (select-project-join-rename under set semantics) as the language for specifying the input query and the reformulations, and constraints expressed as *embedded implicational dependencies* [1, 32]. These include essentially all of the naturally-occurring integrity constraints on relational databases (keys, foreign keys, referential integrity, inverse relationships, functional, join, inclusion and multi-valued dependencies, etc.), and are also ideally suited for capturing physical access paths typically used in query optimization (e.g. materialized views expressed as conjunctive queries, indexes, access support relations, gmaps) [27].

In a nutshell, the $C\&B$ is based on constructing a canonical reformulation called a *universal plan* (because it incorporates redundantly all T-schema elements relevant to the original query), then searching for reformulations among the candidates given by the subqueries of the universal plan. The purpose of the search through the subqueries of the universal plan is to eliminate its redundancy in all possible ways, thus obtaining *minimal* reformulations, i.e. reformulations

containing no elements in the FROM clause (relational atoms) that are redundant for the equivalence to hold under the constraints. The inspected subqueries are checked for equivalence under the constraints to the original query using the classical chase procedure [1], which in essence adds to a query elements that are implied by the constraints. The $C\&B$ was shown in [31] to be complete, that is to return all equivalent minimal reformulations of a query under the given constraints. Unfortunately, its completeness does not scale beyond the low end of the spectrum of practically occurring query and constraint set sizes. The main reason is that, even when there are few actual minimal reformulations for a query, the $C\&B$ inspects a number of candidate subqueries of the universal plan that is often exponential in the size of the query and number of views, thus launching exponentially many chases.

In the work presented hereafter, we revisit the $C\&B$ with the aim of preserving completeness while further achieving practically relevant performance. Our complete query reformulation algorithm, $Prov_{C\&B}$, constructs the same universal plan as the $C\&B$, but employs a novel, much more goal-directed search technique, that inspects up to exponentially fewer candidates than the $C\&B$. This search is based on a novel *Provenance-Aware Chase*, which tracks provenance information that serves for tracing the added query elements back to the universal plan subqueries which are responsible for them being added. This allows $Prov_{C\&B}$ to directly "read off" the minimal reformulations from the result of a *single chase* of the universal plan, saving the exponentially many chases of its subqueries, which the original $C\&B$ would perform. We further show that with the Provenance-Aware Chase, the cost of running a complete search for minimal reformulations can be reduced in practice to a small fraction of typical query execution times, and the benefits are potentially huge in practically relevant settings.

The design of the Provenance-Aware Chase was technically challenging, as it turns out that the standard chase is not well-suited for instrumentation towards tracking the required provenance. Directly instrumenting the standard chase turns out to compromise the soundness of the resulting reformulation algorithm (i.e. it would return non-equivalent reformulations). We thus design the Provenance-Aware Chase based on a different, provenance-agnostic chase flavour, which we call the Conservative Chase and which, as we formally show, is able to provide the required sound behaviour, thus ensuring the overall correctness of $Prov_{C\&B}$.

The remainder of this chapter is organised as follows: we recall the $C\&B$ in Section 1.1, then present a high-level overview and a set of essential intuitions on the $Prov_{C\&B}$ in Section 1.2. We formally describe the $Prov_{C\&B}$ in Section 1.3, and present its theoretical soundness and completeness guarantees, as well as a detailed description of the chase procedures it relies on. We present the implementation of $Prov_{C\&B}$ in Section 1.4 and its evaluation in Section 1.5. We further show how to efficiently adapt $Prov_{C\&B}$ to compute minimum-cost reformulations by introducing incremental cost-based pruning in Section 1.6. We discuss related work in Section 1.7.

1.1 Overview of the Chase & Backchase

We dedicate this section to recalling the $C\&B$ algorithm. We start by recalling the main concepts it relies on:

Queries and subqueries. The $C\&B$ algorithm applies to queries and reformulations expressed as select-project-join-rename (SPJR) queries with set semantics (a.k.a. conjunctive queries). In other words, these are SQL queries (with no nesting and no aggregation) comprising a SELECT DISTINCT clause, a FROM clause and a WHERE clause consisting exclusively of equalities ("=") among column names or between column names and constants, combined using "AND". We refer to the variables in the FROM clause of such query as *tuple variables*. We call *(projection) attributes* the items in the WHERE clause of the form $r.A$, where $R\ r$ is in the FROM clause and A is a column of the table R.

Given a conjunctive query Q as above and a subset of its tuple variables, we will in the following denote by the *subquery* of Q *induced by the given set of tuple variables*, the conjunctive query Q' obtained as follows:

- the FROM clause of Q' contains all the FROM clause elements of Q corresponding to the tuple variables that induce Q'
- the WHERE clause of Q' comprises the (explicit or implicit) equalities in the WHERE clause of Q that use attributes of Q''s FROM clause elements.
- the SELECT DISTINCT clause of Q' contains the same attributes as the SELECT DISTINCT clause of Q, potentially replaced by attributes of Q', such that Q' is syntactically correct and any replacement attribute is in the *same equivalence class* with the original one, according to the reflexive, symmetric and transitive closure of the equalities in the WHERE clause of Q.

Remarks. Note first that several replacements of attributes for the third point above might be possible. The resulting queries being equivalent, we will hereafter refer to *the subquery* induced by a subset of tuple variables. Note also that the third point above cannot be achieved for any subset of the tuple variables of Q. Indeed, the construction of a valid SELECT DISTINCT clause for Q' is achievable iff for any attribute in the SELECT DISTINCT clause of Q, there is at least one other member in its equivalence class such that it is *an attribute of Q''s FROM clause*.

Given a subset of the tuple variables of Q for which one cannot construct a syntactically correct subquery, by a slight abuse of terminology, and to the purpose of ensuring the uniformity of the developments hereafter, we will refer to such non-valid induced subquery as *unsafe*.

Example 1.1.1. *Consider the schema $R(A)$, $S(C, D)$, $T(E)$ and the query:*

Q : *select distinct* $r.A$ *from* $R\ r$, $S\ s$, $T\ t$ *where* $s.C = r.A$ *and* $t.E = s.D$

Then the query:

Q_1 : *select distinct* $s.C$ *from* $S\ s$, $T\ t$ *where* $t.E = s.D$

is a subquery of Q, induced by s and t, and is also a safe *subquery. Note the replacement of $r.A$ by $s.C$ in the SELECT clause, which is possible because of the corresponding equality in the*

WHERE clause of Q.

On the other hand, the subquery Q_2 induced by t alone is unsafe*. Indeed, the only attribute of Q_2, $t.E$, is not equated directly or indirectly to $r.A$ in the WHERE clause of Q.*

Constraints. The $C\&B$ algorithm takes as input constraints expressed as embedded dependencies [1], thus comprising TGDs (tuple generating dependencies) and EGDs (equality generating dependencies), and having the following general form (see Example 1.1.2 for an example of such constraints):

$$\forall r_1, \ldots, r_m, r_1 \in R_1 \wedge \cdots \wedge r_m \in R_m \wedge E_1 \Rightarrow \exists s_1, \ldots, s_n, s_1 \in S_1 \wedge \cdots \wedge s_n \in S_n \wedge E_2$$

where $R_1, \ldots, R_m$, $S_1, \ldots, S_n$ are relations in $S \cup T$, the membership predicates $r_i \in R_i$ paralleling the contents of the FROM clause of an SQL query, and E_1 and E_2 are conjunctions of equalities on the attributes of $r_1 \in R_1, \ldots, r_m \in R_m$, respectively $r_1 \in R_1, \ldots, r_m \in R_m, s_1 \in S_1, \ldots, s_n \in S_n$, paralleling the contents of the WHERE clause of a query. Intuitively, such constraints enforce the fact that if the tuples $r_1, \ldots, r_m$ exist in a database (in the corresponding $R_1, \ldots, R_m$ tables) and respect the conditions of equality in E_1, then the tuples $s_1, \ldots, s_n$ must also exist in the database (in the corresponding tables) and the conditions of E_2 must be verified as well. If the set of tuple variables s_i is empty then the constraint is said to be an EGD (it only enforces equalities on the tuples r_i, as does for instance a key constraint), else the constraint is a TGD. Section 1.3 further provides a detailed description of constraints, their normalized form and their usage throughout our theoretical developments.

Equivalence of queries under constraints. We write $D \models \mathcal{C}$ if a database instance D satisfies all the constraints in a set $\mathcal{C}$. A query Q_1 is contained in query Q_2 under the set $\mathcal{C}$ of constraints (denoted $Q_1 \sqsubseteq_{\mathcal{C}} Q_2$) if and only if $Q_1(D) \subseteq Q_2(D)$ for every database $D \models \mathcal{C}$, where $Q(D)$ denotes the result of Q on D. Q_1 is equivalent to Q_2 under $\mathcal{C}$ (denoted $Q_1 \equiv_{\mathcal{C}} Q_2$) if and only if $Q_1 \sqsubseteq_{\mathcal{C}} Q_2$ and $Q_2 \sqsubseteq_{\mathcal{C}} Q_1$.

Reformulations and minimal reformulations. Let S and T be two relational schemas and $\mathcal{C}$ a set of constraints comprising the relationship between S and T. A *T-reformulation* under $\mathcal{C}$ of a query Q formulated against S (that is, mentioning only relations/tables from S in the FROM clause) is a query R formulated against T, such that $Q \equiv_{\mathcal{C}} R$. A reformulation is *$\mathcal{C}$-minimal* if it contains no elements in the FROM clause that are redundant under the constraints $\mathcal{C}$, i.e. no such element can be removed while preserving equivalence (under $\mathcal{C}$) to the original query.

We further present the $C\&B$ by showing its behaviour on an example [26] of query reformulation. [1].

[1] To ensure readability, the example presents a simple setting of query reformulation, namely that of total rewriting of queries using materialized views, without integrity constraints besides those relating the source and the target schema. Examples including additional integrity constraints are given in Section 1.5.

Example 1.1.2. *Assume that a software company stores some of its internal information in the following schema:*

R(A,B,C), S(C,D), T(D,E).

The R table shows software engineers' assignment to teams, as tuples engineer id(A), engineer role(B), team id(C). One software engineer can participate in several teams and possibly hold several roles in a given team. The S table represents teams' participation on products, as tuples team id(C), product id(D). A team can of course work on several products, and several teams may collaborate on a given product. Finally, the T table lists the high priority production incidents as tuples product id(D), incident id(E).

To achieve rapid incident resolution, the QA manager needs to email all the engineers that could help fix the incidents. The list of these engineers is determined by issuing the following query[2]

Q : *select r.A from R r, S s, T t where r.C=s.C and s.D=t.D,*

Now assume that the following views have been materialized:

V_R*(A,C):* *select r.A, r.C from R r*
V_S*(C,D):* *select s.C, s.D from S s*
V_{RS}*(A,D):* *select r.A, s.D from R r, S s where r.C=s.C*
V_T*(D,E):* *select t.D, t.E from T t*

V_R shows engineers' participation in teams, regardless of their role. V_{RS} lists every engineer's participation on products. It is easy to see that

R_1*: select v_r.A from V_R v_r, V_S v_s, V_T v_t where v_r.C=v_s.C and v_s.D=v_t.D*
R_2*: select v_{rs}.A from V_{RS} v_{rs}, V_T v_t where v_{rs}.D=v_t.D*

are equivalent rewritings of Q using the views (these are total rewritings, as they use no base schema tables). Also, each rewriting is minimal, in the sense that none of their FROM clause elements can be removed while preserving equivalence to Q. Note that given a choice of such FROM clause elements, the equalities among their attributes are uniquely determined for the resulting query to be a reformulation.

The $C\&B$ algorithm analyses the above problem as an instance of the reformulation problem, for which the source schema is the schema against which the query Q is formulated (tables R, S, and T in this example), and the target schema is the schema of the materialized views (tables V_R, V_S, V_{RS} and V_T). The set $\mathcal{C}$ of dependencies relating the two schemas is obtained by unioning the set $\mathcal{C}_\mathcal{I}$ of integrity constraints (empty in our example) with the set $\mathcal{C}_\mathcal{V}$ of embedded dependencies expressing the set $\mathcal{V}$ of view definitions. These embedded dependencies are all TGDs and are presented below. For each of the view definitions, two TGDs are produced, a

[2]Since all queries in this paper are interpreted under set semantics, we systematically drop the DISTINCT keyword for conciseness.

forward one (denoted by the letter c) and a backward one (denoted by the letter b). Note that the backward constraints are not full TGDs, that is, they present, right of the implication arrow, attributes that are undetermined, for instance the $r.B$ attribute in the case of b_{V_R}.

$$
\begin{array}{ll}
c_{V_R}\text{: } \forall r, r \in R & \rightarrow \exists v_r, v_r \in V_R \wedge v_r.A = r.A \wedge v_r.C = r.C \\
b_{V_R}\text{: } \forall v_r, v_r \in V_R & \rightarrow \exists r, r \in R \wedge r.A = v_r.A \wedge r.C = v_r.C \\
c_{V_S}\text{: } \forall s, s \in S & \rightarrow \exists v_s, v_s \in V_S \wedge v_s.C = s.C \wedge v_s.D = s.D \\
b_{V_S}\text{: } \forall v_s, v_s \in V_S & \rightarrow \exists s, s \in S \wedge s.C = v_s.C \wedge s.D = v_s.D \\
c_{V_{RS}}\text{:} \forall r, s,\ r \in R \wedge s \in S \wedge r.C = s.C & \rightarrow \exists v_{rs}, v_{rs} \in V_{RS} \wedge v_{rs}.A = r.A \wedge v_{rs}.D = s.D \\
b_{V_{RS}}\text{:} \forall v_{rs}, v_{rs} \in V_{RS} & \rightarrow \exists r, s,\ r \in R \wedge s \in S \wedge r.A = v_{rs}.A \\
 & \quad \wedge s.D = v_{rs}.D \wedge r.C = s.C \\
c_{V_T}\text{: } \forall t, t \in T & \rightarrow \exists v_t, v_t \in V_T \wedge v_t.D = t.D \wedge v_t.E = t.E \\
b_{V_T}\text{: } \forall v_t, v_t \in V_T & \rightarrow \exists t, t \in T \wedge t.D = v_t.D \wedge t.E = v_t.E
\end{array}
$$

The $C\&B$ algorithm relies on the *chase* procedure, which essentially adds to a query the redundant elements implied by the constraints. This is accomplished by repeatedly applying a syntactic transformation called a *chase step*. To describe it, we introduce some terminology. We call *relational atoms* the membership predicates occurring in the constraints (e.g. $r \in R$ in b_{V_R} in Example 1.1.2) and use the same name for the variable bindings occurring in the FROM clause of a query (e.g. $R\ r$ in query Q in Example 1.1.2) because they express the same concept with different syntax. We call *equality atoms* the equalities occurring in constraints or the WHERE clause of a query. The *premise* of a constraint is the set of atoms left of the implication arrow, while the conclusion is the set of atoms to its right.

The chase step checks if the premise d_P of a constraint $d \in \mathcal{C}$ matches into the query, in which case the query is extended with atoms constructed from the conclusion d_C. The match is a function h from the premise variables to the query variables, which maps the premise atoms into query atoms. This function is known as a homomorphism [17]. The extension of the query involves adding to the FROM clause the relational atoms from d_C (with fresh variable names to avoid clashes with existing variables in the FROM clause) and to the WHERE clause the equalities from d_C (occurrences of premise variables are replaced by their image under h). If the standard chase considers that these atoms *already exist* in the query (i.e. a homomorphism extension exists), then the chase step is said to *not apply*, and it turns into a no-op[3].

Example 1.1.3. *We illustrate a chase step of query Q from Example 1.1.2 with constraint $c_{V_{RS}}$. The identity mapping on the premise variables matches the relational atoms $r \in R$ and $s \in S$ and the equality atom $r.C$=$s.C$ into, respectively, the first and second relational atoms in Q's FROM clause and the first equality atom in its WHERE clause. The chase step adds the conclusion atoms to Q, yielding:*

[3]In general, when chasing with EGDs, there may exist a third case, besides application and non-application: a chase step with an EGD may *fail* if it equates explicitly or implicitly *two distinct constants*. We consider in the following only input comprising a to-be-reformulated query and a set of constraints that are compatible, that is, such failing of a chase step may not occur. Alternatively, when one failing step is encountered in a chase sequence in the $C\&B$, one could conclude directly to the non-existence of reformulations.

Q' : *select* $r.A$
from $R\ r$, $S\ s$, $T\ t$, $V_{RS}\ v_{rs}$
where $r.C{=}s.C$ *and* $s.D{=}t.D$ *and* $v_{rs}.A{=}r.A$ *and* $v_{rs}.D{=}s.D$.

The result of chasing a query Q with a set of constraints $\mathcal{C}$ is obtained by applying a sequence of chase steps until the query can be no longer extended. We denote this result with $Q^{\mathcal{C}}$.[4] The $C\&B$ algorithm proceeds in two phases:

1. **Chase:** The input query Q is chased with the constraints $\mathcal{C}$, to obtain a chase result $Q^{\mathcal{C}}$. Next, the *universal plan* U is constructed by restricting $Q^{\mathcal{C}}$ to schema T, i.e. by keeping only T-elements in the FROM clause and the corresponding equalities in the WHERE clause.

2. **Backchase:** This phase checks the subqueries of the universal plan U for equivalence (under $\mathcal{C}$) to Q. The equivalence check is performed according to a classical result [1]: it involves chasing each subquery sq and checking that Q has a containment mapping into $sq^{\mathcal{C}}$. A subquery is in the result set of the $C\&B$ if it respects this equivalence check and furthermore it is minimal.

Example 1.1.4. The chase phase. *When chasing Q with $\mathcal{C} = \mathcal{C}_{\mathcal{V}}$, the only chase steps that apply involve $c_{V_R}, c_{V_S}, c_{V_T}, c_{V_{RS}}$, yielding the chase result:*

$Q^{\mathcal{C}_{\mathcal{V}}}$*: select r.A*
from *R r, S s, T t,* $V_R\ v_r$, $V_S\ v_s$, $V_T\ v_t$, $V_{RS}\ v_{rs}$
where r.C=s.C and s.D=t.D and v_r*.A=r.A and* v_r*.C = r.C and* v_s*.C=s.C*
and v_s*.D=s.D and* v_t*.D=t.D and* v_t*.E=t.E and* v_{rs}*.A=r.A and* v_{rs}*.D=s.D*

Restricting $Q^{\mathcal{C}_{\mathcal{V}}}$ *to the view schema yields the universal plan*[5]*:*

U*: select* v_r*.A*
from $V_R\ v_r$, $V_S\ v_s$, $V_T\ v_t$, $V_{RS}\ v_{rs}$
where v_r*.C=*v_s*.C and* v_s*.D=*v_t*.D and* v_r*.A=*v_{rs}*.A and* v_s*.D=*v_{rs}*.D*

The backchase phase. *In this phase, the subqueries of U are inspected. Notice that R_1, R_2 above are among them, being induced by the sets of tuple variables $\{v_r, v_s, v_t\}$, respectively $\{v_{rs}, v_t\}$. To find out that R_2 is equivalent to Q, the $C\&B$ first chases R_2 with $\mathcal{C}_{\mathcal{V}}$. The only applicable chase steps involve $b_{V_{RS}}, b_{V_T}$, yielding the result:*

[4] While the chase is not guaranteed to terminate in general, we confine ourselves here to terminating chases, which yield a finite result. It is well-known that the resulting query is not necessarily unique, as it depends on the non-deterministic choices made during the chase sequence among simultaneously applicable chase steps. However, the result is unique up to equivalence [1], which suffices for our purposes. We will therefore refer to "the" chase result hereafter.

[5] Equalities of terms involving view variables that were implicit in $Q^{\mathcal{C}_{\mathcal{V}}}$ are made explicit in U, by taking the transitive closure of the equality relation.

$R_2^{\mathcal{C}v}$: *select* v_{rs}.A
from V_{RS} v_{rs}, V_T v_t, R r, S s, T t
where v_{rs}.D=v_t.D *and* r.A=v_{rs}.A *and* s.D=v_{rs}.D *and* s.C= r.C *and* t.D=v_t.D *and* t.E=v_t.E

Since the identity mapping on variables is a containment mapping from Q to $R_2^{\mathcal{C}v}$, R_2 is equivalent to Q, and thus a rewriting. R_2 is moreover minimal, since none of its subqueries is a rewriting of Q (the backchase checks this by trying the subqueries). R_2 is therefore output by the $C\&B$ algorithm. R_1 is discovered analogously. It turns out that there are no other minimal rewritings of Q. The backchase phase determines this by systematically checking the other subqueries of U, but discarding them as not being equivalent to Q, or not being minimal. For instance, the subquery

sq: *select* v_r.A *from* V_R v_r, V_T v_t

is not a rewriting of Q because equivalence doesn't hold, and the subquery

sq': *select* v_{rs}.A *from* V_{RS} v_{rs}, V_S v_s, V_T v_t
where v_{rs}.D=v_s.D *and* v_s.D=v_t.D

is a rewriting but is not minimal, since by removing the atom V_S v_s from the FROM clause one obtains R_2 which is itself a rewriting, therefore the V_S v_s is redundant, unnecessary for equivalence.

The fact that rewritings R_1 and R_2 in Example 1.1.2 are discovered among the subqueries of U is not accidental. In [27], it was shown that all minimal rewritings of Q are (isomorphic to) subqueries of U, and this result was extended to the presence of integrity constraints expressed as embedded dependencies, as long as they ensure terminating chases, in [31].

Note that, as further emphasized in Section 1.3, for arbitrary sets $\mathcal{C}$ of constraints, the chase procedure is not guaranteed to terminate. One of the least restrictive and most referenced conditions on $\mathcal{C}$ known to date, that is sufficient to ensure chase termination regardless of the input query Q, is called *weak acyclicity* [33].

Practical performance of the $C\&B$. [60] describes the first $C\&B$ implementation and identifies the backchase phase as the practical performance bottleneck; this is due to the **exponentially many subqueries of the universal plan that are chased** so as to be checked for equivalence with the original query. To improve performance, the follow-up work then proposes techniques for pruning the search, the only completeness-preserving pruning technique being the one sketched in [60] and detailed in [59]. This technique boils down to simply enumerating subqueries of the universal plan U *in a bottom-up fashion*, starting with all single-atom subqueries, next with two-atom subqueries, etc, pruning thus subqueries that are known to be not minimal because they already include a minimal reformulation (such as sq' in Example 1.1.4).

The pruning achieved by the above strategy, in turn, although beneficial, **still does not avoid the chase of a potentially exponential number of subqueries** (obviously, at least all those subqueries with less relational atoms then the smallest minimal reformulation will be chased).

Moreover, if no reformulations exist, then no pruning can be applied and all the subqueries of the universal plan are chased. To avoid the unnecessary effort in this case, [60] proposes to first check that a rewriting exists. This check is based on the property that a reformulation exists iff the universal plan is itself a reformulation - that is, if Q has a containment mapping into the result of chasing U with $\mathcal{C}$.

Example 1.1.5. *Continuing with Example 1.1.4, a possible chase sequence of U with $\mathcal{C}_\mathcal{V}$ involves, in order, chase steps with $b_{V_{RS}}$, b_{V_R}, b_{V_S} and b_{V_T}, yielding*

$U^{\mathcal{C}_\mathcal{V}}$*: select* v_r.A
from $V_R\ v_r$, $V_S\ v_s$, $V_T\ v_t$, V_{RS}, v_{rs}, R r_1, S s_1, R r_2, S s_2, T t
where v_r.C=v_s.C *and* v_s.D=v_t.D *and* v_r.A=v_{rs}.A *and* v_s.D=v_{rs}.D *and* r_1.A=v_{rs}.A
and s_1.D=v_{rs}.D *and* r_1.C=s_1.C *and* r_2.A=v_r.A *and* r_2.C=v_r.C
and s_2.C=v_s.C *and* s_2.D=v_s.D *and* t.D = v_t.D *and* t.E = v_t.E

There exist two containment mappings from Q to $U^{\mathcal{C}_\mathcal{V}}$, namely $h_1 = \{r \mapsto r_1, s \mapsto s_1, t \mapsto t\}$ and $h_2 = \{r \mapsto r_2, s \mapsto s_2, t \mapsto t\}$. Therefore at least one rewriting exists, and one can further examine the universal plan subqueries in search of minimal rewritings.

Even with the above improvement, the bottom-up search strategy **fails to achieve practically relevant performance**, due essentially to the same problem of a large number of chases. Moreover, and unfortunately, among those subquery chases that are in cause for the decrease in performance, many turn out to be *fruitless*, because after chasing no containment mapping is found, hence the subquery is (expensively) chased only to be discarded by the absence of a mapping. Moreover, one can note a high degree of *redundant chasing*, of the atoms and groups of atoms occurring in common within distinct subqueries. By construction of the $C\&B$, this redundancy cannot be avoided.

Example 1.1.6. *In Example 1.1.2, the bottom-up search strategy will prune the superqueries of R_1, R_2, i.e the subqueries induced by v_r, v_{rs}, v_t and v_s, v_{rs}, v_t, as well as the universal plan itself. Unsafe subqueries (e.g. those induced by v_s, v_t, and v_s, v_t) will also be pruned, since only safe rewritings are of interest. However, the following 7 subqueries of U (induced by): v_r; v_{rs}, v_r, v_s; v_r, v_t; v_r, v_{rs}; v_s, v_{rs}; v_r, v_s, v_{rs} will all be fruitlessly chased, only to discover that they are not rewritings of Q. Furthermore, the atom $V_T\ v_t$ is redundantly chased multiple times (with U to determine the existence of a rewriting, with R_1 and R_2 etc.).*

1.2 A novel algorithm: the Provenance-Aware Chase & Backchase

We dedicate this section to showing a different and much more efficient approach of the backchase phase, and to presenting a high level overview of the resulting novel reformulation algorithm, the Provenance-Aware Chase & Backchase.

Indeed, we will sketch in the following (and demonstrate with our experimental evaluation) how, by our new strategy, the performance of a complete search for minimal reformulations can be significantly more improved than just by the naive bottom-up strategy and the corresponding pruning. The essential way of achieving such performance improvement is that of **replacing**

the potentially exponential number of subquery chases with a single chase of the universal plan.

To ensure a sound and complete reformulation algorithm, this single chase should in turn be able to retain all the relevant effect of the individual subquery chases. To achieve such behaviour, we will thus instrument this chase with *provenance annotations*, whose final purpose will be to *reflect the minimal reformulations*, that is, the subqueries of the universal plan that turn out to be (minimally) equivalent to Q. The ability to maintain and propagate in an unexpensive fashion such provenance information during a single chase of the universal plan, would then *spare the exponentially many chases of its subqueries*, which constitute the performance issue in the $C\&B$. By design, such approach will also avoid *the fruitlessness and redundancy in subquery chases*.

By attaching provenance annotations to the atoms added during the chase, our goal will be to identify, for each added atom, the parts of the universal plan (the original T-schema atoms in the universal plan) that are responsible for creating the atom. Our annotations will then further allow, once the annotated chase has finished, *the minimal reformulations to be directly "read-off"*, by putting together individual atom annotations, and thus obtaining the required minimal subqueries. The following example sketches the intuition behind this approach:

Example 1.2.1. *We revisit Example 1.1.5 and show again the atoms resulting from the chase of the universal plan, this time adding their corresponding provenance annotations in square brackets. The view atoms originally in the universal plan are annotated with their corresponding (unique) tuple variables. The atoms corresponding to relations R, S and T are annotated according to the view atom that, by means of a corresponding chase step, was responsible for introducing them:*

$U^{\mathcal{C}_\mathcal{V}}$: *select* v_r.A
from $V_R\ v_r[v_r]$, $V_S\ v_s[v_s]$, $V_T\ v_t[v_t]$, $V_{RS}\ v_{rs}[v_{rs}]$,
$R\ r_1[v_{rs}]$, $S\ s_1[v_{rs}]$, $R\ r_2[v_r]$, $S\ s_2[v_s]$, $T\ t[v_t]$
where v_r.C=v_s.C *and* v_s.D=v_t.D *and* v_r.A=v_{rs}.A *and* v_s.D=v_{rs}.D *and* r_1.A=v_{rs}.A
and s_1.D=v_{rs}.D *and* r_1.C=s_1.C *and* r_2.A=v_r.A *and* r_2.C=v_r.C
and s_2.C=v_s.C *and* s_2.D=v_s.D *and* t.D = v_t.D *and* t.E = v_t.E

Recall the two containment mappings that we have shown from Q to $U^{\mathcal{C}_\mathcal{V}}$, h_1 comprising r_1, s_1 and t, and h_2, comprising r_2, s_2 and t. The provenance annotations and these mappings then allow reading off minimal reformulations as follows: the first image of Q puts together the annotations v_{rs} (two times, redundantly) and v_t. Note how these correspond to the tuple variables inducing the rewriting R_2. The second mapping provides the rewriting R_1 as the tuple variables v_r and v_s and v_t.

Note how a single chase of the universal plan followed by finding the containment mappings from Q to $U^{\mathcal{C}_\mathcal{V}}$ (steps that are already carried out in any efficient implementation of the original $C\&B$, to verify the existence of a reformulation) have allowed us to directly read-off all the minimal reformulations available.

Furthermore, we have avoided the chases of these reformulations, as well as the fruitless chases of the 7 subqueries listed in Example 1.1.6. Note also how the $V_T\ v_t$ atom has been chased

only once, producing the $T\ t[v_t]$ atom whose provenance is then read through the mappings. Finally, note how provenance annotations are simply copied from the premise to the added conclusion.

Provenance-Aware Chase (pa-chase). The idea of the developments hereafter is thus the replacement of the large number of isolated chases of the subqueries of the universal plan with a single chase, which captures via *provenance* the $C\&B$-relevant effect of the isolated chases of the U-subqueries. As sketched in Example 1.2.1, the pa-chase starts by annotating each original relational atom of universal plan U with a *provenance term* corresponding to the tuple variable of this atom, and thus *uniquely identifying each original relational atom* in the universal plan.

Every atom introduced during the pa-chase is further annotated with a *provenance formula*. Provenance formulae are DNF boolean formulae, constructed from provenance terms using logical conjunction and disjunction. Indeed, recall that a subquery of the universal plan is uniquely induced by a subset of the relational atoms of the universal plan. These atoms correspond in turn to provenance terms. A provenance formula in the form of a conjunction of terms then specifies the unique subquery of the universal plan that is induced by these terms. A disjunction expresses alternative such subqueries leading to the construction of the given atom.

Once the universal plan U is pa-chased into result U', to find minimal reformulations, we first compute the set $\mathcal{H}$ of all containment mappings from Q to U'. For each containment mapping, we further compute the provenance formula of its image, which is defined as the conjunction of the individual atoms' formulae. We then produce the DNF form of the disjunction of the formulae corresponding to images of these mappings.

Example 1.2.2. *Recall the result of the pa-chase of the universal plan in Example 1.2.1, and note the provenance terms.*

$U^{\mathcal{C}_V}$*: select* v_r.A
from $V_R\ v_r[v_r]$, $V_S\ v_s[v_s]$, $V_T\ v_t[v_t]$, $V_{RS}\ v_{rs}[v_{rs}]$,
$R\ r_1[v_{rs}]$, $S\ s_1[v_{rs}]$, $R\ r_2[v_r]$, $S\ s_2[v_s]$, $T\ t[v_t]$
where v_r.C=v_s.C *and* v_s.D=v_t.D *and* v_r.A=v_{rs}.A *and* v_s.D=v_{rs}.D *and* r_1.A=v_{rs}.A
and s_1.D=v_{rs}.D *and* r_1.C=s_1.C *and* r_2.A=v_r.A *and* r_2.C=v_r.C
and s_2.C=v_s.C *and* s_2.D=v_s.D *and* t.D = v_t.D *and* t.E = v_t.E

Recall also the two containment mappings that we have shown from Q to $U^{\mathcal{C}_V}$, h_1 comprising r_1, s_1 and t, and h_2, comprising r_2, s_2 and t. The provenance formula of the image of the first mapping is then $v_{rs} \wedge v_{rs} \wedge v_t$ which simplifies to $v_{rs} \wedge v_t$. The provenance formula of the image of the second mapping is $v_r \wedge v_s \wedge v_t$. The global DNF formula Π representing minimal reformulations is then $(v_{rs} \wedge v_t) \vee (v_r \wedge v_s \wedge v_t)$

While the examples above show our global approach, we develop hereafter the ideas behind the pa-chase, and reveal the complexity of the problem of maintaining sound provenance annotations. Indeed, our original motivation in designing the pa-chase was that of directly achieving, by instrumenting the standard chase with provenance, the following goal:

(†) *The provenance of an atom constructed during the pa-chase of the universal plan specifies the set of minimal U-subqueries whose standard chases (conducted in isolation from each other) would construct the atom.*

The benefits of such a design would be that, as sketched in the example above, (i) by restricting attention to only those universal plan subqueries identified by the provenance annotations we do not miss any minimal reformulations, thus preserving completeness; and (ii) there is no need to further chase the provenance-identified subqueries to check their equivalence to the original query, thus rendering a single chase of U sufficient. This in turn is expected to provide a significant performance improvement over the original $C\&B$, due to the replacement of the expensive backchase phase by a sensibly lighter-weight procedure.

The technical challenge facing the implementation of this idea is raised by the need to carefully instrument the chase procedure to correctly track provenance according to our initial design goal. As detailed shortly, it turns out that as defined the standard chase is not suited for such direct instrumentation. The main reason for this lack of compatibility is that, intuitively, the standard chase is too *aggressive* in the application of its steps, and it will *mix up* atoms that should be kept distinct in order to ensure that their respective provenance corresponds to goal (†). This particular problematic behaviour of the standard chase is linked, as we show hereafter, to the presence of constraints whose conclusion has *undetermined attributes*. Standard chasing with such constraints introduces atoms that are wrongly considered as identical and that, as detailed hereafter, should be kept distinct to ensure the soundness of provenance annotations.

To account for such constraints, we have designed a *less aggressive, more conservative* chase variation, which we call the Conservative Chase. It is provenance-agnostic like the standard chase and essentially equivalent to it in terms of the produced result, and its termination is guaranteed under *weakly acyclic constraints*. On the other hand, the important advantage of the Conservative Chase is that it lends itself to direct provenance-tracking instrumentation *for all the types of embedded dependencies*, yielding the pa-chase which is guaranteed to satisfy the following invariant:

(⋄) *The provenance of an atom constructed during the pa-chase of the universal plan specifies the set of minimal U-subqueries whose Conservative Chases (conducted in isolation from each other) would construct the atom.*

Invariant (⋄) will ensure that in the final DNF formula Π thus obtained, every conjunct is a reformulation. While in our simple example it is further the case that these conjuncts are directly *minimal* reformulations, in the general case we still need to *minimize* the resulting reformulations. In general, given a reformulation R of Q under $\mathcal{C}$, minimizing R would involve searching for its subqueries that are still equivalent to Q (which in turn would be checked by chasing them with $\mathcal{C}$). Once again we employ provenance to avoid chasing. To this end, we observe that conjunct $c_1 \in \Pi$ induces a non-minimal U-subquery if and only if there is a conjunct $c_2 \in \Pi$ that *subsumes* c_1 in the standard Boolean logic sense: c_2's terms are a subset of c_1's (otherwise said, c_1 *implies* c_2). All we need to do therefore is to remove from Π all subsumed conjuncts, obtaining what we call the *reduced form* of Π, $rf(\Pi)$. The conjuncts of $rf(\Pi)$ each induce minimal reformulations. Notice that this minimization not only avoids chasing, but it avoids even the

construction of reformulations, involving instead only lightweight manipulations of provenance conjuncts.

We give a simplified, high level view of $Prov_{C\&B}$ below:

Provenance-Aware Chase & Backchase (source schema S, target schema T, set of weakly acyclic constraints $\mathcal{C}$, query Q)

//chase phase:

1. compute universal plan U
   by standard-chasing Q with $\mathcal{C}$ and keeping only T-atoms

//provenance-directed reformulation search:

2. compute the result U' of pa-chasing U with $\mathcal{C}$
3. compute the set $\mathcal{H}$ of all containment mappings from Q into U'
4. compute Π as the DNF formula of $\bigvee_{h\in\mathcal{H}} \pi(h(Q))$, for $\pi(h(Q))$ the formula of the image of h
5. compute the reduced form $rf(\Pi)$ of Π
6. return the U-subqueries induced by the conjuncts of $rf(\Pi)$.

Note that, while the requirement of weak acylicity for the input constraints is stated specifically above, the $Prov_{C\&B}$ will be sound and complete in general for sets of constraints for which *both the standard chase and the Conservative Chase are guaranteed to terminate*. While the characterization of such (more complex than weak acyclicity) conditions is beyond the scope of this work, it is certainly a very interesting follow-up direction, as we underline in our conclusions chapter. We also emphasize there the interest of further refining the complexity analysis, both time and space-wise, for the Provenance-Aware Chase. Indeed, compared to a non-annotated chase version, the Provenance-Aware Chase could possibly introduce a *significant space overhead*, by the worst case exponential space complexity for the provenance formulae. While we show with our experimental section that our algorithm exhibits a satisfactory behaviour even in stress-test practical scenarios, there are an important number of optimization directions worth exploring in order to further improve the efficiency of our approach.

We present in Section 1.3 the detailed description of $Prov_{C\&B}$ and its formal guarantees: namely, we show that $Prov_{C\&B}$ is **sound and complete**, thus returning *all* and *precisely* the minimal reformulations of Q. We dedicate the remainder of this section to further exploring the main intuitions behind the central brick of the $Prov_{C\&B}$ algorithm, the pa-chase.

Details on pa-chase. The design of the pa-chase walks a fine line between tracking provenance as desired and ensuring termination of the resulting chase. We detail below some of the intuitions and analysis that led to its design. Hereafter, we will denote the provenance formula of an atom a by $\pi(a)$. The provenance formula of a set of atoms A (as shown above for the image of a mapping) is obtained as the logical conjunction of the provenance formulae of its members: $\pi(A) = \bigwedge_{a\in A} \pi(a)$.

Recall that our first approach of the design of the pa-chase was an attempt to mimic the behaviour of the standard chase, by "plugging in" directly provenance annotations. Accordingly,

we present the intuitions below as referring to a *tentative pa-chase step (tpa-step)*, modelled after the standard chase step. In due course, we identify the need to substitute the standard chase with the Conservative Chase in the actual definition of a pa-chase step, and Goal (†) with Invariant ($\diamond$).

I1: the provenance of the image of the premise is transferred to the atoms introduced by the chase step. Assume that a sequence of pa-chase steps has yielded a result q. Assume that a standard chase step s with dependency d using match h applies on q, adding a set A of atoms to q. By definition of the standard chase step, the premise d_P therefore has an image $h(d_P)$ in q. By Goal (†), the U-subqueries whose chases in isolation create this image are indicated by $\pi(h(d_P))$. Since each of these chases creates $h(d_P)$ in isolation, they each can be extended with chase step s, so each of the U-subqueries in $\pi(h(d_P))$ when standard-chased in isolation construct the atoms in A. To record this fact, the tpa-step adds the A-atoms and annotates each of them with $\pi(h(d_P))$. For instance, in Example 1.2.1, the pa-chase step with $b_{V_{RS}}$ matches the premise against the relational atom $V_{RS}\ v_{rs}$, and it introduces the relational and equality atoms involving tuple variables r_1, s_1 (shown in $U^{\mathcal{C}_\mathcal{V}}$), annotating relational atoms with provenance v_{rs}.

Towards ensuring termination, the standard chase never applies a step if it attempts to add atoms that are *already there* (the step turns into a no-op). The notion of being "already there" is formalized in the standard chase to mean that the premise's homomorphic match h compatibly extends to a homomorphic match of the conclusion. Denoting the extended match as h', the atoms in q that are "already there" are then the atoms in $h'(d_C)$. In designing the pa-chase step, one would be then tempted to parallel the standard chase step, turning the former step into a no-op in this case. It turns out however that the pa-chase step must diverge from its standard counterpart.

I2: when the same atom a can be introduced by chasing several alternative U-subqueries, a's provenance must reflect this. Consider first the case when the atoms that are "already there" are identical copies of the set A of atoms the standard chase step (with constraint d, using premise match h) would attempt to add. Note that when adding relational atoms, the standard chase step invents fresh names for the tuple variables, so when referring to an atom $a \in A$ as an identical copy of an atom $c \in h'(d_C)$, we mean that all their attributes are pairwise equal. Recall from case *I1* that $\pi(a) = \pi(h(d_P))$. Now if $\pi(a)$ contains at least one U-subquery sq that is not in $\pi(c)$, then the isolated chase of sq would never construct c, hence the standard chase step constructing a would apply. In view of Goal (†), the tpa-step records this behaviour by extending the provenance formula of c with a disjunction with $\pi(h(d_P))$. We call such a step *provenance-enriching* because instead of creating new atoms it only enriches the provenance of existing ones.

I3: if the chase step produces atoms that match into q without being identical copies of the match image, these atoms must be added and their provenance recorded. The technically most subtle case for defining the pa-chase step is the one in which the atoms that the standard chase step attempts to add (A) are not identical to those that are "already there" ($h'(d_C)$).

Example 1.2.3. *Recall the pa-chase of universal plan U from Example 1.2.1, and assume that this time the first two chase steps applied involve first b_{V_R}, then b_{V_S} (the standard chase selects randomly among the applicable steps, so we can observe a chase sequence distinct from the one in Example 1.2.1). The intermediate result is U_2 below, in which the tuple variables are named to show correspondence to the tuple variables introduced in Example 1.2.1.*

U_2: *select* $v_r.A$
from $V_R\ v_r$, $V_S\ v_s$, $V_T\ v_t$, $V_{RS}\ v_{rs}$, $R\ r_2[v_r]$, $S\ s_2[v_s]$
where $v_r.C{=}v_s.C$ *and* $v_s.D{=}v_t.D$ *and* $v_r.A{=}v_{rs}.A$ *and* $v_s.D{=}v_{rs}.D$
and $r_2.A{=}v_r.A$ *and* $r_2.C{=}v_r.C$ *and* $s_2.C{=}v_s.C$ *and* $s_2.D{=}v_s.D$

Now consider a tpa-chase step with $b_{V_{RS}}$ on U_2 as defined above. The standard chase step would attempt to add the relational atoms $R\ r_1$, $S\ s_1$ as well as all equalities they are involved in (these can be seen in $U^{\mathcal{C}_\mathcal{V}}$ in Example 1.2.1). However the standard chase step would not apply, as there is a match of r_1, s_1 into r_2, s_2 respectively, which matches the equality atoms involving r_1, s_1 into the (explicit or implicit) equality atoms involving r_2, s_2. Notice on the other hand that r_2 is not *a copy of r_1; indeed, the equality $r_1.B{=}r_2.B$ does not follow, because the constraint $b_{V_{RS}}$ leaves the B attribute undetermined ($b_{V_{RS}}$ is not a full TGD).*

Where can we then record provenance information of these new, distinct atoms? The intuition offered by the standard chase, illustrated above, would be to add no new atoms, because they are "already there" in the form of $h'(d_C)$. If we were to follow this intuition, then the natural way to record the newly discovered provenance would be to enrich the provenance of the atoms in $h'(d_C)$, paralleling intuition *I2*. This would be wrong however, as the U-subqueries in $\pi(h(d_P))$ are only known to cause the construction of the atoms in A and not of the distinct ones in $h'(d_C)$. The following example shows that for a pa-chase step defined in this way, the resulting provenance of the atoms in $h'(d_C)$ would spuriously contain U-subqueries whose standard chase does not actually construct them:

Example 1.2.4. *Consider the relational schema R(A), S(B, C, D), and the following query and set of views:*

Q : *select* $r.A$ *from* $R\ r, S\ s$ *where* $s.B = r.A$ *and* $s.C = 1$ *and* $s.D = 2$
$V_1(A)$: *select* $r.A$ *from* $R\ r, S\ s$ *where* $s.B = r.A$ *and* $s.C = 1$ *and* $s.D = 2$
$V_2(A)$: *select* $r.A$ *from* $R\ r, S\ s$ *where* $s.B = r.A$ *and* $s.C = 1$

In the corresponding total-view based rewriting problem, the universal plan is then as follows (with initial provenance annotations shown in square brackets):

U : *select* $v_1.A$ *from* $V_1\ v_1[v_1], V_2\ v_2[v_2]$ *where* $v_1.A = v_2.A$

Now assume that we enrich the standard chase with direct provenance instrumentation and consider the chase sequence with the backwards constraints corresponding to V_1 and V_2. The first chase step with the constraint corresponding to V_1:

$$\forall v_1, v_1 \in V_1 \longrightarrow \exists r, s,\ r \in R \wedge s \in S \wedge r.A = v_1.A \wedge s.B = r.A \wedge s.C = 1 \wedge s.D = 2$$

leads to the corresponding provenance-annotated result:

U' : *select* $v_1.A$ *from* $V_1\ v_1[v_1], V_2\ v_2[v_2], R\ r_1[v_1], S\ s_1[v_1]$
where $v_1.A = v_2.A$ *and* $r_1.A = v_1.A$ *and* $s_1.B = r_1.A$ *and* $s_1.C = 1$ *and* $s_1.D = 2$

. *A standard chase step with the backward constraint corresponding to* V_2*:*

$$\forall v_2, v_2 \in V_2 \longrightarrow \exists r, s,\ r \in R \wedge s \in S \wedge r.A = v_2.A \wedge s.B = r.A \wedge s.C = 1$$

would then not apply*, because the standard chase would consider the atoms to be introduced as being* already there*. If we were then to record the provenance of these atoms by following the standard chase, the provenance-annotated result of the chase step would be:*

U'' : *select* $v_1.A$ *from* $V_1\ v_1[v_1], V_2\ v_2[v_2], R\ r_1[v_1 + v_2], S\ s_1[v_1 + v_2]$
where $v_1.A = v_2.A$ *and* $r_1.A = v_1.A$ *and* $s_1.B = r_1.A$ *and* $s_1.C = 1$ *and* $s_1.D = 2$

In other words, this step would simply enrich *the provenance of the* r_1 *and* s_1 *atoms. But the* s_1 *atom above* cannot be constructed using only the subquery induced by v_2*, that is* $V_2\ v_2$*, because* V_2 *does not operate any selection on the* D *attribute! Accordingly, the provenance formula* $v_1 + v_2$ *of* s_1 *is incorrect.*

Still, the provenance of the non-identical atoms that would be introduced by the pa-chase step *has to be recorded somewhere*, to ensure completeness of the reformulations. The tpa-chase chase step should then be *allowed to add these atoms*, and in this respect behave *more conservatively* then the standard chase. For the example 1.2.3, the tpa-chase step is therefore allowed to add to U_2 the new atoms r_1 and s_1 resulting from the chase with $b_{V_{RS}}$, adorning them with v_{rs}, and thus as a final result obtaining the same pa-chased universal plan $U^{\mathcal{C}_\mathcal{V}}$ as in Example 1.2.1:

$U^{\mathcal{C}_\mathcal{V}}$: select v_r.A
from $V_R\ v_r[v_r]$, $V_S\ v_s[v_s]$, $V_T\ v_t[v_t]$, $V_{RS}\ v_{rs}[v_{rs}]$,
R $r_1[v_{rs}]$, S $s_1[v_{rs}]$, R $r_2[v_r]$, S $s_2[v_s]$, T $t[v_t]$
where v_r.C=v_s.C and v_s.D=v_t.D and v_r.A=v_{rs}.A and v_s.D=v_{rs}.D and r_1.A=v_{rs}.A
and s_1.D=v_{rs}.D and r_1.C=s_1.C and r_2.A=v_r.A and r_2.C=v_r.C
and s_2.C=v_s.C and s_2.D=v_s.D and t.D = v_t.D and t.E = v_t.E

I4: disallow the infinite reapplication of the same chase step. As seen in Examples 1.2.3 and 1.2.4, case *I3* occurs when at least one of the relational atoms in the conclusion of a constraint d has some undetermined attribute. *Undetermined attributes* are involved neither directly nor indirectly in equalities with attributes of the relational atoms in the premise d_P, and therefore their value is *not determined by the match of* d_P. For instance, attribute B of tuple variable r is undetermined in both b_{V_R} and $b_{V_{RS}}$.

While the tentative definition of the pa-chase step according to case *I3* above would introduce *distinct atoms* and thus keep track of provenance as desired, its divergence from the standard chase step would immediately lead to *non-termination* due to same chase step now applying

infinitely often. Indeed, in Example 1.2.3 above, the tpa-chase step with $b_{V_{RS}}$ is allowed to introduce tuple variables r_1, s_1 and their atoms A despite their match into r_2, s_2 and their atoms, because for example r_1.B=r_2.B does not hold. But then the same tpa-chase step can apply again, introducing fresh tuple variables r_1', s_1' and atoms A', which match into r_1, s_1 and A without being identical copies, because r_1'.B=r_1.B does not hold.

To disallow infinitely many reapplications of a chase step with the same constraint d and premise match h, we normalize d to turn all undetermined attributes in its conclusion into determined attributes. We employ a classical technique from First-Order Logic, namely normalization by equating the undetermined attributes with function calls, corresponding to the classical *Skolem functions* one would obtain when eliminating existential quantifiers from the constraints if written in First-Order Logic form.

Function symbols used in calls must be distinct across constraints (so that the chase step with a constraint is not mistaken for a reapplication of a chase step with a distinct constraint). While intuitively function calls should take as arguments all tuple variables of the premise, it turns out that (as presented in Section 1.3) to soundly distinguish between non-identical atoms for provenance bookkeeping purposes, it is sufficient to consider fewer function arguments: namely, those attributes of the premise tuples that also appear in (the equalities of) the conclusion. By this procedure, the attributes that were undetermined in the original form of the dependencies become now determined by the Skolem terms, in short *Skolem-determined*.

Example 1.2.5. *We illustrate only for constraint b_{V_R}, whose normalization involves setting the undetermined attribute r.B equal to a function call:*
$\forall v_r, v_r \in V_R \rightarrow \exists r, r \in R \wedge r.A = v_r.A \wedge r.C = v_r.C \wedge r.B = f(v_r.A,\ v_r.C)$

The Conservative Chase. We call the provenance-unaware chase flavour conservatively enforcing atom identity as above the *Conservative Chase*. As detailed in Section 1.3, when checking whether an atom with a Skolem-determined attribute is "already there", the Conservative Chase step requires an identical copy thereof in q, such that in this copy Skolem function calls only match calls with the same function symbol and pairwise identical arguments. We will show in Section 1.3 that the Conservative Chase is essentially equivalent to the standard chase in terms of its result, thus ensuring invariant ($\diamond$) is equivalent to ensuring goal ($\dagger$). As we will show, the Conservative Chase has the central benefit of being able to provide soundness for the provenance annotations and the corresponding reformulations *for all embedded dependencies*.

Revisiting cases *I1* through *I3*, which prescribe the behaviour of the tentative tpa-chase step, we adjust this design by making the pa-chase record the provenance of atoms constructed by the Conservative Chase instead of the standard chase. More specifically, in the description of the tpa-chase step in cases *I1* through *I3* above, the standard chase step with dependency d is replaced by a Conservative Chase chase step with the Skolemized version of d, denoted $sk(d)$. We detail extensively this construction in the following section.

1.3 Formal presentation and guarantees of $Prov_{C\&B}$

We have provided in the previous section a high-level overview of the $Prov_{C\&B}$, together with a number of informal details and essential intuitions regarding its global flow. This section will be

dedicated to a formal description of the $Prov_{C\&B}$ and the concepts it relies on (such as atoms, chase procedures and provenance formulae), as well as to providing its theoretical guarantees.

We start by briefly reviewing, in Subsection 1.3.1, a set of basic notions informally introduced in previous sections. We continue by formally describing the chase procedure in Subsection 1.3.2. We then present, in Subsection 1.3.3, the Conservative Chase – which, as mentioned in previous sections, is the chase flavor designed to be compatible with provenance annotations. Based on the Conservative Chase we introduce, in Subsection 1.3.4, the Provenance-Aware Chase, which is the essential brick of the $Prov_{C\&B}$. Finally, in Subsection 1.3.5, we give a detailed description of the $Prov_{C\&B}$ algorithm and show that it is sound and complete, that is, it finds *all* and *precisely* the minimal reformulations of the input query.

1.3.1 Preliminaries: atoms, queries and constraints

Let $\mathcal{R}$ be a relational schema and $\mathcal{K}$ a set of constants.

Relational atoms. A *relational atom* over $\mathcal{R}$ is a predicate of the form $r \in R$, where R is a relation in $\mathcal{R}$ and r is called a *tuple variable*. A $valid\ set$ of relational atoms over $\mathcal{R}$ is a set of relational atoms over $\mathcal{R}$, $\{r_1 \in R_1, \ldots, r_n \in R_n\}$, such that for $i \neq j$, $r_i \neq r_j$, that is, all tuple variables are distinct.

Projection terms. If S is a $valid\ set$ of relational atoms over $\mathcal{R}$, we denote by the *projection terms* of S the set $ProjTerms(S)$=$\{r_i.A_j\}$, where $(r_i \in R_i)$ is in S and A_j is an attribute of R_i.

Equality atoms. An *equality atom* over a set A is an equality $t_1 = t_2$ such that $t_i, t_j \in A$. For E a set of equality atoms over A, we denote by $Clos(E)$ the *reflexive, symmetric, and transitive* closure of E. For $A' \subseteq A$ a subset of A, we define the *restriction of E to A'* as the subset of E, $E' = E|_{A'}$, such that $(t_1 = t_2) \in E'$ iff $(t_1 = t_2) \in E$ and $t_1, t_2 \in A'$.

Constraints. We consider *constraints* over $\mathcal{R}$ and $\mathcal{K}$ expressed as logical implications in the form:

$$\forall r_1, \ldots, r_m, r_1 \in R_1 \wedge \cdots \wedge r_m \in R_m \wedge E_1 \Rightarrow \exists s_1, \ldots, s_n, s_1 \in S_1 \wedge \cdots \wedge s_n \in S_n \wedge E_2, \quad (1.1)$$

where $\{r_1 \in R_1, \ldots, r_m \in R_m, s_1 \in S_1, \ldots, s_n \in S_n\}$ is a $valid\ set$ of relational atoms over $\mathcal{R}$, E_1 is a conjunction of equality atoms over $ProjTerms(\{r_1 \in R_1, \ldots, r_m \in R_m\}) \cup \mathcal{K}$ and E_2 is a conjunction of equality atoms over $ProjTerms(\{r_1 \in R_1, \ldots, r_m \in R_m\}) \cup ProjTerms(\{s_1 \in S_1, \ldots, s_m \in S_m\}) \cup \mathcal{K}$.

Queries. In the following, we will use the term *queries* to denote standard Select-From-Where (SFW) queries over $\mathcal{R}$ and $\mathcal{K}$, with set semantics (for conciseness we will omit the DISTINCT keyword, but it is always implied).

1.3.2 The Standard Chase

We will dedicate this subsection to the description of the chase procedure, hereafter called the Standard Chase. As mentioned in the previous sections, the Standard Chase is an iterative procedure consisting in a sequence of steps. To express the Standard Chase, we will in the following introduce the concept of $body$, and show how queries and constraints can be expressed using $bodies$.

1.3.2.1 Bodies

Definition 1.3.1 ($body$)**.** *A $body$ B over a relational schema $\mathcal{R}$ and a set of constants $\mathcal{K}$ consists in:*

1. *a $valid\ set$ of relational atoms over $\mathcal{R}$ which we denote by $[B]_{rel}$*
2. *a set of equality atoms over $ProjTerms([B]_{rel}) \cup \mathcal{K}$, which we denote by $[B]_{eq}$.*

We denote by the *tuple variables* of B the set $TupVar(B) = \{r_i\}$, s.t. $(r_i \in R_i)$ is in $[B]_{rel}$. We denote by the *terms* of B the set $T(B) = ProjTerms([B]_{rel}) \cup \mathcal{K}$. We further distinguish the *instantiated* terms of B, as those terms in $T(B)$ that appear in equalities in $[B]_{eq}$.

Closed version of a $body$. In the following, reasoning about equivalence concerning $bodies$ will *always* be based on their *closed* versions. We define the closed version of a $body$ B as the $body$ $\overline{B}$, such that $[\overline{B}]_{rel} = [B]_{rel}$ and $[\overline{B}]_{eq} = Clos([B]_{eq})$. We say that a $body$ is *closed* if $B = \overline{B}$.

Example 1.3.2. *Consider the relational schema $\mathcal{R}$ = {R(A, B), S(C)} and the set of constants $\mathcal{K}$ = {1}.*

Then B_1 = {$r \in R$, $s \in S$, $r.A = 1$, $r.B = s.C$} is a body over $\mathcal{R}$ and $\mathcal{K}$, with $[B_1]_{rel}$ = {$r \in R$, $s \in S$}, $[B_1]_{eq}$ = {$r.A = 1$, $r.B = s.C$}, $T(B_1)$={r.A, r.B, s.C, 1}, $TupVar(B_1)$={r, s} and $\overline{B_1}$ = {$r \in R$, $s \in S$, $r.A = r.A$, $r.B = r.B$, $s.C = s.C$, $1 = 1$, $r.A = 1$, $1 = r.A$, $r.B = s.C$, $s.C = r.B$}

However B_2 = {$r \in R$, $r \in S$} is not a body over $\mathcal{R}$ and $\mathcal{K}$, because $[B_2]_{rel}$={$r \in R$, $r \in S$} is not a $valid\ set$ of relational atoms.

Also, B_3 = {$r \in R$, $r.X = 1$} is not a body over $\mathcal{R}$ and $\mathcal{K}$ because $r.X = 1$ is not an equality atom over $ProjTerms([B_3]_{rel}) \cup \mathcal{K}$.

Given two $bodies$ B_1 and B_2, we write $B_1 \subseteq B_2$ to denote the fact that $[B_1]_{eq} \subseteq [B_2]_{eq}$ and $[B_1]_{rel} \subseteq [B_2]_{rel}$. We define the *restriction* of a $body$ B to a sub-schema $\mathcal{R}' \subseteq \mathcal{R}$ as the maximal $body$ $B' \subseteq B$ such that all relational atoms of B' are over $\mathcal{R}'$.

We define the union, intersection and difference of two $bodies$ as a set of relational atoms and a set of equality atoms obtained by pairwise union, intersection and difference of their corresponding relational and equality atoms. Note that the results of such operations *are not necessarily $bodies$*. However, we extend the notion of inclusion above to such results.

In the following, unless explicitly specified otherwise, we will consider queries, constraints and $bodies$ over a fixed relational schema $\mathcal{R}$ and a fixed set of constants $\mathcal{K}$; we will thus hereafter

consider these parameters common and implicit in the subsequent definitions and theoretical results.

Constraints expressed with bodies. We associate to a constraint C of form (1.1), two *bodies* defined as follows:

- a body C_{prem}, called the *premise* of C, such that:
  1. $[C_{prem}]_{rel}$ = $\{r_1 \in R_1, \ldots, r_m \in R_m\}$
  2. $[C_{prem}]_{eq}$ = $\{eq\}$, s.t. eq in E_1.
- a body C_{concl}, called the *conclusion* of C, such that:
  1. $[C_{concl}]_{rel}$ = $\{r_1 \in R_1, \ldots, r_m \in R_m, s_1 \in S_1, \ldots, s_n \in S_n\}$
  2. $[C_{concl}]_{eq}$ = $\{eq\}$, s.t. eq in E_2.

Example 1.3.3. *Consider the constraint $c_{V_{RS}}$ in Example 1.1.2:*
$c_{V_{RS}} : \forall r, s,\ r \in R \wedge\ s \in S \wedge\ r.C = s.C \rightarrow\ \exists v_{rs}, v_{rs} \in V_{RS} \wedge v_{rs}.A = r.A \wedge\ v_{rs}.D = s.D$
Then $c_{V_{RS}\,prem}$=$\{r \in R,\ s \in S,\ r.C = s.C\}$
and $c_{V_{RS}\,concl}$=$\{r \in R,\ s \in S,\ v_{rs} \in V_{RS},\ v_{rs}.A = r.A,\ v_{rs}.D = s.D\}$

Note that for a given constraint C, the couple (C_{prem}, C_{concl}) allows a straightforward and completely determined reconstruction of the form (1.1) of the constraint. In the following we will thus refer without ambiguity to the constraint C as the couple of *bodies* (C_{prem}, C_{concl}).

Normalized form of constraints Without loss of generality, we assume that every constraint C has the following normalized form:

- if C_{concl} has no other relational atoms besides those of C_{prem}, then it contains a single equality atom $t_1 = t_2$, such that $t_1, t_2 \in T(C_{prem})$ and we say that C is an *equality generating dependency* (EGD).
- otherwise, all equalities in $[C_{concl}]_{eq}$ have at least one term in $T(C_{concl})$ - $T(C_{prem})$ and are:
  - of the form $s_i.A$ = constant, if there is a constant in the equivalence class of $s_i.A$, as induced by $Clos([C_{prem}]_{eq} \cup [C_{concl}]_{eq})$, or else
  - of the form $s_i.A$ = projection term in $T(C_{prem})$, if there is a premise projection term in the equivalence class of $s_i.A$, or else
  - of the form $s_i.A = s_j.B$, if there is no projection premise term and no constant in the equivalence class of $s_i.A$.

  Moreover, if t_1 and t_2 are two distinct premise terms occurring in $[C_{concl}]_{eq}$, then the equality $t_1 = t_2$ is not in $Clos([C_{prem}]_{eq})$.

  In this case we say that C is a *tuple generating dependency* (TGD).

Distinguished premise terms. Given a constraint C (in the normalized form above), we define the *distinguished premise terms* of C, denoted by $DTPrem(C)$, as the set of projection terms of C_{prem} that appear in the equalities of C_{concl}.

Example 1.3.4. *Consider again the constraint $c_{V_{RS}}$ in Example 1.1.2,*
$c_{V_{RS}} : \forall r, s,\ r \in R \wedge\ s \in S \wedge\ r.C = s.C \rightarrow\ \exists v_{rs}, v_{rs} \in V_{RS} \wedge v_{rs}.A = r.A \wedge\ v_{rs}.D = s.D$

We have seen that $c_{V_{RS}\,prem}$={$r \in R,\ s \in S,\ r.C = s.C$} and $c_{V_{RS}\,concl}$={$r \in R,\ s \in S,\ v_{rs} \in V_{RS},\ v_{rs}.A = r.A,\ v_{rs}.D = s.D$}.

$c_{V_{RS}}$ is then a TGD and the set of distinguished premise terms of $c_{V_{RS}}$ is $DTPrem(c_{V_{RS}})$ = {$r.A,\ s.D$}.

By the above definition of the normalized form of constraints, it follows directly that for a constraint C, $C_{concl} \cap C_{prem} = [C_{prem}]_{rel}$.

Queries expressed with bodies. For a query Q, we denote by *body*(Q) the *body* B such that:

1. $[B]_{rel}$ = {$r_i \in R_i$} s.t. $R_i\ r_i$ is in the FROM clause of Q.
2. $[B]_{eq}$ = {eq}, such that eq is in the WHERE clause of Q.

Example 1.3.5. *Consider the query Q in Example 1.1.2:*
$Q : select\ r.A\ from\ R\ r,\ S\ s,\ T\ t\ where\ r.C = s.C\ and\ s.D = t.D$
Then B = body(Q) = {$r \in R$, $s \in S$, $t \in T$, $r.C = s.C$, $s.D = t.D$},
with $[B]_{rel}$ = {$r \in R$, $s \in S$, $t \in T$}
and $[B]_{eq}$ = {$r.C = s.C$, $s.D = t.D$}.

It is easy to show that for a syntactically correct SFW query Q, *body*(Q) is indeed a *body*. However, in the case of queries, *body*(Q) does not allow reconstructing Q without ambiguity since it is obvious that we miss the projection attributes of Q. This missing information can be retrieved if we further associate to Q a subset of $ProjTerms([body(Q)]_{rel})$, denoted by *Head*(Q), and obtained by copying all projection attributes in the SELECT clause of Q.

Given a *body* B and a subset H of $ProjTerms([B]_{rel})$, we denote by *Query*(H, B) the SFW query "reconstructed" in an unambiguous fashion from H and B.

1.3.2.2 Homomorphisms of bodies

We will characterize the Standard Chase by means of *homomorphisms* of *bodies*:

Definition 1.3.6 (*homomorphisms* of *bodies*)**.** *Let h be a function from the tuple variables of body B_1 to the tuple variables of body B_2. Based on h we can define the following two additional functions:*

1. *a function h_{terms} over $T(B_1)$ such that:*
   - *$h_{terms}(r.A)$ = $h(r).A$, for $r.A$ a projection term in $T(B_1)$*
   - *$h_{terms}(K)$ = K, for K a constant term in $T(B_1)$*
2. *a function h_{atoms} over $\overline{B_1}$, such that:*

- $h_{atoms}(r \in R) = (h(r) \in R)$, *for* $(r \in R)$ *in* $[B_1]_{rel}$
- $h_{atoms}(t_1 = t_2)$ = $(h_{terms}(t_1) = h_{terms}(t_2))$, *for* $(t_1 = t_2)$ *in* $[\overline{B_1}]_{eq}$.

We say that h *is a* $homomorphism$ *iff:*

1. *for each relational atom* a *in* $[B_1]_{rel}$, $h_{atoms}(a)$ *is in* $[B_2]_{rel}$.
2. *for each equality atom* a *in* $[B_1]_{eq}$, $h_{atoms}(a)$ *is in* $[B_2]_{eq}$.

For a function h defined on the tuple variables of a $body$ B_1, since h completely determines h_{terms} and h_{atoms}, in the following, to avoid clutter, we will use the notation h to refer to h_{terms} and h_{atoms} whenever the domain of application is clear. In particular, for any set of atoms $S \subseteq \overline{B_1}$ (even if S is not a $body$), we will use the notation $h(S)$=$\{h_{atoms}(a)\}$, $a \in S$.

One can show that the composition of two $homomorphisms$ is also a $homomorphism$. If there exists h $homomorphism$ from B_1 to B_2 and h' $homomorphism$ from B_2 to B_1, they are said to be *homomorphically equivalent*. If there exists h a $homomorphism$ from B_1 to B_2, and furthermore h is bijective (on the tuple variables) and h^{-1} is a $homomorphism$ from B_2 to B_1, we call h an *isomorphism* and B_1 and B_2 are said to be isomorphic. Note that if two $bodies$ are isomorphic they are of course homomorphically equivalent.

Compatible $homomorphisms$. Let h be a $homomorphism$ from a $body$ B_1 to a $body$ B. Let h' be a $homomorphism$ from a $body$ B_2 to a $body$ B'. We say that h and h' are *compatible* if $h' = h$ on $TupVar(B_1) \cap TupVar(B_2)$.

Containment and equivalence of queries through $homomorphisms$ *of* $bodies$. Given the way of obtaining $bodies$ from queries, it is easy to show that the following holds:

Proposition 1.3.7. *Let* Q_1 *and* Q_2 *be two queries with the same SELECT clause. Then:*

1. $Q_1 \sqsubseteq Q_2$ *iff there exists a* $homomorphism$ *from* $\overline{body(Q_2)}$ *to* $\overline{body(Q_1)}$.
2. Q_1 *and* Q_2 *are equivalent iff* $\overline{body(Q_1)}$ *and* $\overline{body(Q_2)}$ *are homomorphically equivalent.*

1.3.2.3 Standard Chase steps and sequences

We are now ready to formally define the Standard Chase steps, using the notions of $bodies$ and $homomorphisms$ of $bodies$.

As will be the case for all the other chase flavours presented throughout this paper, we present Standard Chase steps by first listing their *conditions of application* and then by specifying their *application*, i.e. how their output is constructed from the input.

Definition 1.3.8 (Standard Chase step conditions of application)**.** *A standard chase step with constraint* C *on a body* B *applies iff:*

1. *there exists a* $homomorphism$ h *from* C_{prem} *to* $\overline{B}$,
2. *there exists no* $homomorphism$ h' *compatible with* h, *from* C_{concl} *to* $\overline{B}$.

Definition 1.3.9 (Standard Chase step application)**.** *The application of a Standard Chase step with constraint C, on a body B, given homomorphism h from C_{prem} to $\overline{B}$, results into a new body, B' = $chase\_step\_res(B, C, h)$ such that $B' \supset B$ and B' is obtained from B as follows:*

1. *let $B' = B$*
2. *add to B' the relational atoms $s'_1 \in S_1, \ldots, s'_n \in S_n$ (if any), using fresh tuple variables (one for each relational atom specific to C_{concl})*
3. *define the function h' from the tuple variables of C_{concl} into B' such that*
   (a) *$h'(r) = h(r)$ for each tuple variable r in $C_{prem} \cap C_{concl}$*
   (b) *$h'(s_j) = s'_j$, for each remaining tuple variable s_j in C_{concl}*
4. *for each equality atom eq in $[C_{concl}]_{eq}$, add to B' the equality atom $h'(eq)$*

Example 1.3.10. *Let B = {$r \in R$, $s \in S$, $t \in T$, $r.C = s.C$} (the body corresponding to the query Q from Example 1.1.2).*

Let $C = (C_{prem}, C_{concl})$, where C_{prem}={$r \in R$, $s \in S$, $r.C = s.C$} and C_{concl}={$r \in R$, $s \in S$, $v_{rs} \in V_{RS}$, $v_{rs}.A = r.A$, $v_{rs}.D = s.D$} (the constraint $c_{V_{RS}}$ from Example 1.1.2, expressed as a couple of bodies).

There exists a homomorphism h from C_{prem} to $\overline{B}$, such that $h(r) = r$, $h(s) = s$ and $h((r.C = s.C)) = (r.C = s.C)$.

However, there exists no homomorphism compatible with h from C_{concl} to $\overline{B}$ (no relational atom $v'_{rs} \in V_{RS}$ exists in B).

Thus, the Standard Chase step with C given h applies on B, yielding B' = {$r \in R$, $s \in S$, $t \in T$, $r.C = s.C$, $v'_{rs} \in V_{RS}$, $v'_{rs}.A = r.A$, $v'_{rs}.D = s.D$}.

It is easy to show that the function h' constructed in the Standard Chase step application on a *body* B, given a constraint C and a *homomorphism* h from C_{prem} to $\overline{B}$, is a *homomorphism* compatible with h, from C_{concl} to B'. We will hereafter call h' the *Standard Chase step compatible homomorphism* for the given step.

Standard Chase sequences. Given a *body* B and a set of constraints $\mathcal{C}$, a Standard Chase sequence consists in producing the *bodies* B_0, B_1, …, such that:

1. B_0 = B
2. B_i is obtained by B_{i-1} by the following operations:
   (a) pick $C \in \mathcal{C}$ s.t. a Standard Chase step with C applies on B_{i-1}, with a *homomorphism* h from C_{prem} to $\overline{B_{i-1}}$;
   (b) let B_i:=$chase\_step\_res(B_{i-1}, C, h)$;

For a finite Standard Chase sequence with a number of steps k, we denote by the *result* of the sequence the body B_k produced by the last step.

A *full* Standard Chase sequence consists in applying Standard Chase steps as long as there exist at least a constraint such that a Standard Chase step applies. A *terminating* Standard Chase sequence is a full Standard Chase sequence that terminates after a finite number of steps n - that is, there exists B_n such that for any constraint in $\mathcal{C}$, and any possible $homomorphism\ h$ from C_{prem} to $\overline{B_n}$, there exists a compatible $homomorphism$ from C_{concl} to $\overline{B_n}$.

The Standard Chase of queries. We extend the notion of Standard Chase step in a straightforward fashion to queries: a Standard Chase step with constraint C applies on a query Q iff there exists a $homomorphism\ h$ from C_{prem} to $\overline{Body(Q)}$. The result of applying such chase step on a query is the query $Q' = Query(Head(Q), chase\_step\_res(Body(Q), C, h))$.

We can further generalize the notion of Standard Chase sequences to queries. It is easy to show that given a query Q, the result Q' of a Standard Chase sequence of Q with $\mathcal{C}$ is $Q' = Query(Head(Q), B')$, where B' is the result of the corresponding Standard Chase sequence on $Body(Q)$.

Given this direct correspondence, we will in the following, unless explicitly stated otherwise, refer to the Standard Chase as the Standard Chase of $bodies$. We will refer to the corresponding queries in the few specific cases where we wish to emphasize this correspondence.

1.3.2.4 Properties of the Standard Chase

We conclude our presentation of the Standard Chase by reminding two known results from the literature (mentioned briefly in previous sections) regarding the Standard Chase termination and results.

We have seen that full Standard Chase sequences may not terminate. One of the least restrictive and most referenced conditions concerning the termination of *all* full Standard Chase sequences over a given input is stated in Theorem 3.9 in [33], and relies on a property known as the *weak acyclicity* of a set of constraints.

We remind here the result from [33] regarding weakly acyclic constraints:

Theorem 1.3.11. *Let B be a body and $\mathcal{C}$ a set of weakly acyclic constraints.*

Then there exists a polynomial in the size of B that bounds the length of every full Standard Chase sequence of B with $\mathcal{C}$. In particular, all such sequences terminate.

Moreoever, note that the choice of steps in a Standard Chase sequence is non-deterministic. One can thus produce, starting from a given $body$, full Standard Chase sequences with different behaviour, in terms of termination or results. It has been shown however that, on a given input, even when not all full Standard Chase sequences are guaranteed to terminate, any two terminating sequences lead to equivalent results, as follows:

Theorem 1.3.12. *Let B be a body and $\mathcal{C}$ a set of constraints. Let B_1 and B_2 be the results of two terminating Standard Chase sequences with $\mathcal{C}$ over B.*

Then $\overline{B_1}$ and $\overline{B_2}$ are homomorphically equivalent.

1.3.3 The Conservative Chase

We have noted in Section 1.2 the fact that the Standard Chase does not directly lend itself to provenance annotations, creating the need for the design of a different chase flavour, which we call the Conservative Chase. While Section 1.2 gives an initial overview and a set of essential intuitions, this section will focus on the formal presentation of the Conservative Chase, hereafter denoted the cs_chase.

The cs_chase is, similar to the Standard Chase, an iterative procedure consisting in a sequence of steps. In the case of the cs_chase however, these steps will be based on a generalization of $bodies$ which we will call sk_bodies. Furthermore, the cs_chase steps will employ a different form of constraints, hereafter called $sk\_constraints$.

1.3.3.1 Skolem terms, sk_bodies and sk_constraints

To describe sk_bodies and $sk\_constraints$, we first introduce the concept of Skolem terms.

Definition 1.3.13 (Skolem terms)**.** *Let $\mathcal{F}$ be a set of function symbols of fixed arity. We define recursively the set of Skolem terms induced by $\mathcal{F}$ over a set S, denoted by $SkTerms(S, \mathcal{F})$, as the set $\{f_i(a_i^1, \ldots, a_i^{n_i})\}$, where $f_i \in \mathcal{F}$, n_i is the arity of f_i, and $(a_i^1, \ldots, a_i^{n_i})$ is an ordered subset (potentially empty) of $S \cup SkTerms(S, \mathcal{F})$.*

Example 1.3.14. *Let $S = \{x\}$ and $\mathcal{F} = \{f\}$, where the arity of f is 2. Then $f(x, x)$, $f(f(x, x), x)$, $f(f(x, f(x, x)), x)$ are some of the Skolem terms in $SkTerms(S, \mathcal{F})$.*

We define sk_bodies as a generalization of $bodies$. This generalization mainly consists in the addition of a special set of equalities called *constructive equalities*. Constructive equalities will in turn be expressed over projection terms, constants, and *Skolem terms*, as follows:

Definition 1.3.15 (sk_body)**.** *An sk_body B over a relational schema $\mathcal{R}$, a set of constants $\mathcal{K}$ and a set of function symbols of fixed arity $\mathcal{F}$ consists in:*

1. *a $valid\ set$ of relational atoms over $\mathcal{R}$ which we denote by $[B]_{rel}$*
2. *a set of equality atoms over $ProjTerms([B]_{rel}) \cup \mathcal{K}$, which we denote by $[B]_{eq}$*
3. *a set of equality atoms over $ProjTerms([B]_{rel}) \cup \mathcal{K} \cup SkTerms(ProjTerms([B]_{rel}), \mathcal{F})$, which we denote by $[B]_{constr\_eq}$ and call the* constructive equalities *of B.*

   The constructive equalities have the property that there exists a subset S of $[B]_{rel}$, such that:

   (a) every constructive equality in $[B]_{constr\_eq}$ is of the form $t = t'$,
   where $t \in ProjTerms([B]_{rel} - S)$
   and $t' \in ProjTerms(S) \cup SkTerms(ProjTerms(S), \mathcal{F}) \cup \mathcal{K}$.

   (b) every projection term t in $ProjTerms([B]_{rel} - S)$ participates in exactly one constructive equality *(of the form above).*

Note that by the definition above the subset S is unique. For an sk_body we will denote such subset by $[B]_{b-rel}$.

Example 1.3.16. *Consider the relational schema $\mathcal{R}$ = {R(A), S(B, C)}, the set of constants $\mathcal{K}$ = {1} and the set of function symbols $\mathcal{F}$={f_1,f_2}.*

Then B = {$r \in R$, $s \in S$, $s.B = f_1(r.A)$, $s.C = f_2(r.A)$, $s.C = 1$}, $[B]_{constr\_eq}$={$s.B = f_1(r.A)$, $s.C = f_2(r.A)$}, $[B]_{eq}$ = {$s.C = 1$} is an sk_body over $\mathcal{R}$, $\mathcal{K}$ and $\mathcal{F}$, where $[B]_{b-rel}$ = {$r \in R$}

Sk_bodies as a generalization of bodies. We have mentioned that $bodies$ are a sub-class of sk_bodies. Indeed, a $body$ B is a special type of sk_body where $[B]_{b-rel}$=$[B]_{rel}$ and $[B]_{constr\_eq}$ = ϕ. To underline the fact that an sk_body is also a $body$, we will employ the term *regular* $body$.

Intuitively, the constructive equalities in an sk_body B will hold the "history of construction" of the projection terms in B during the Conservative Chase. We will be particularly interested in Conservative Chase sequences that start from a $body$. Not surprisingly, the $[B]_{b-rel}$ part of any sk_body B thus obtained will consist in the relational atoms of the initial $body$. Every term that is "further added" by the Conservative Chase will be connected to this initial $body$ by means of the constructive equalities.

As we did for $bodies$, we denote by the *tuple variables* of B the set $TupVar(B)$ = {r_i}, s.t. ($r_i \in R_i$) is in $[B]_{rel}$.

We further denote by the *terms* of B the set $T(B)$ = $ProjTerms([B]_{rel}) \cup \mathcal{K} \cup SkTerms(ProjTerms([B]_{rel}), \mathcal{F})$. We distinguish the set of *instantiated* terms of B as those terms occurring in the equalities of B. Note that not all Skolem terms of B are instantiated: indeed, according to the definition of an sk_body, only Skolem terms in $SkTerms(ProjTerms([B]_{b-rel}), \mathcal{F})$ will appear in the (constructive) equalities of B.

As we did for $bodies$, we will in the following assume a fixed relational schema, set of constants *and set of function symbols $\mathcal{F}$, such that moreover $\mathcal{F}$ contains an infinite number of symbols for any given arity.*

Constructive terms. Based on the constructive equalities of an sk_body B, we can associate to every term t in $T(B)$ its *constructive term*, $ConstrT(t)$, which is a term in $ProjTerms([B]_{b-rel})$ $\cup$ $SkTerms(ProjTerms([B]_{b-rel}), \mathcal{F}) \cup \mathcal{K}$ as follows:

1. if t is a constant, $ConstrT(t)$ = t
2. if t is a projection term in $ProjTerms([B]_{b-rel})$, $ConstrT(t)$ = t
3. if t is a projection term in $ProjTerms([B]_{rel} - [B]_{b-rel})$, $ConstrT(t)$ = t' where $t = t'$ is the unique constructive equality involving t in $[B]_{constr\_eq}$
4. if t a Skolem term with no argument $f()$, $ConstrT(t)$ = t
5. if t is a Skolem term of the form $f(a_1, \ldots, a_n)$, $ConstrT(t)$ = $f(ConstrT(a_1), \ldots, ConstrT(a_n))$.

Example 1.3.17. *Consider the sk_body B = {$r \in R$, $s \in S$, $s.B = f_1(r.A)$, $s.C = f_2(r.A)$}, with $[B]_{constr\_eq}$={$s.B = f_1(r.A)$, $s.C = f_2(r.A)$}*

Then:

- $ConstrT(r.A)$ = $r.A$
- $ConstrT(s.B)$ = $f_1(r.A)$
- $ConstrT(s.C)$ = $f_2(r.A)$
- $ConstrT(f_3(s.C))$ = $f_3(f_2(r.A))$

Note that according to the definition of constructive terms, *every term in* $T(B)$, be it instantiated or not, has its associated constructive term. Note moreover that if B is a $body$, then $ConstrT(t)$ = t for all terms t of B.

Relational atom identity by constructive terms. Collapsible atoms. For two sk_bodies B_1 and B_2 (not necessarily distinct), we further introduce the concept of *collapsible atoms*. Two relational atoms $(r_1 \in R) \in [B_1]_{rel}$ and $(r_2 \in R) \in [B_2]_{rel}$ are collapsible if for each of their projection terms, $ConstrT(r_1.A_j)$ = $ConstrT(r_2.A_j)$. In other words, all their pairwise projection terms have *identical constructive terms.*

Example 1.3.18. *Consider again the sk_body B = {$r \in R$, $s \in S$, $s.B = f_1(r.A)$, $s.C = f_2(r.A)$}, with $[B]_{constr\_eq}$={$s.B = f_1(r.A)$, $s.C = f_2(r.A)$}.*

Let B_1 = {$r \in R$, $t \in T$, $s_1 \in S$, $s_1.B = f_1(r.A)$, $s_1.C = f_2(r.A)$} with $[B_1]_{constr\_eq}$={$s_1.B = f_1(r.A)$, $s_1.C = f_2(r.A)$}.

Then the $(s \in S)$ atom in B and the $(s_1 \in S)$ atom in B_1 are collapsible.

Recall that, when providing the initial intuitions on the Conservative Chase, we have mentioned the need of (conservatively) enforcing atom identity. Intuitively, two relational atoms will be considered as "identical" by the cs_chase when they are collapsible.

Closed version of an sk_body. As is the case for $bodies$, reasoning about equivalence in terms of sk_bodies will also be based on their *closed* version. We define the closed version of an sk_body B as the sk_body $\overline{B}$ such that:

1. $[\overline{B}]_{rel}$ = $[B]_{rel}$
2. $[\overline{B}]_{eq}$ = $Clos([B]_{constr\_eq} \cup [B]_{eq})|_{ProjTerms(B)}$
3. $[\overline{B}]_{constr\_eq}$ = $[B]_{constr\_eq}$

Example 1.3.19. *Let B be an sk_body such that $[B]_{rel}$={$r \in R$, $t \in T$, $s \in S$}, $[B]_{constr\_eq}$ = {$s.B = f(r.A)$, $s.C = f(r.A)$} and $[B]_{eq}$ = $t.D = r.A$. Note that $[B]_{b-rel}$ = {$r \in R$, $t \in T$}*

Then $[\overline{B}]_{eq}$ = {$s.B = s.C$, $t.D = r.A$} (and of course all the symmetric of these and the reflexive equalities).

On the other hand, if, instead of the above, $[B]_{constr\_eq}$ = {$s.B = r.A$, $s.C = f(r.A)$}, then $[\overline{B}]_{eq}$ = {$s.B = r.A$, $s.B = t.D$, $t.D = r.A$} (and of course all the symmetric of these and the reflexive equalities).

Note that the regular equalities in the closed version of an sk_body allow "reconstructing" all possible equalities between projection terms, mixing constructive equalities and regular ones in the given sk_body. The constructive Skolem terms only participate as a transitivity element in the above computation.

Note also that if B contains no constructive equalities (and is thus a $body$), then the definition of the closed version of B corresponds to the definition of the closed version previously defined for $bodies$, thus ensuring the correctness of our notation.

Bodies from sk_bodies. While all $bodies$ are also sk_bodies, the reverse is however not true. We associate to an sk_body B a canonical $body$ denoted by $Body(B)$ and constructed by removing constructive equalities from B:

1. $[Body(B)]_{rel} = [B]_{rel}$
2. $[Body(B)]_{eq} = [B]_{eq}$

Sk_constraints. We have mentioned that the cs_chase steps use a different expression of constraints, called $sk\_constraints$. We define $sk\_constraints$ as follows:

Definition 1.3.20 (Sk_constraints)**.** *An $sk\_constraint$ C is a couple of sk_bodies (C_{prem}, C_{concl}), called the premise and the conclusion of C, such that:*

1. *the premise of C is a regular $body$ ($[C_{prem}]_{constr\_eq} = \emptyset$),*
2. $[C_{prem}]_{rel} \subseteq [C_{concl}]_{rel}$
3. $[C_{concl}]_{b-rel} = [C_{prem}]_{rel}$.

Furthermore:

1. *if C_{concl} has no other relational atoms besides those of C_{prem}, then it is a regular $body$ containing a single equality atom $eq = (t_1 = t_2)$, $eq \in [C_{concl}]_{eq}$, $[C_{concl}]_{constr\_eq} = \phi$ and we say that C is an* sk equality generating dependency*, sk_EGD.*
2. *otherwise, the conclusion of C has only constructive equalities ($[C_{concl}]_{eq} = \phi$) and there exists a subset of the projection terms of $[C_{prem}]_{rel}$, called the* distinguished premise terms*, $DTPrem(C) = \{a_1, \ldots, a_n\}$, such that for $i \neq j$, $(a_i = a_j)$ is not in $Clos([C_{prem}]_{eq})$, and for every constructive equality $t = t'$ in the conclusion, t' is either*

   (a) *a constant*

   (b) *a distinguished premise term a_i*

   (c) *a Skolem term of the form $f(a_1, \ldots, a_n)$*

   In this case we say that C is a sk tuple generating dependency*, sk_TGD.*

Given the definition of an $sk\_constraint$, it is easy to show that $C_{concl} \cap C_{prem} = [C_{prem}]_{rel}$.

Skolem-determined terms of an sk_TGD. We denote by a *Skolem-determined term* in the conclusion of an sk_TGD C a projection term t of $[C_{concl}]_{rel}$ - $[C_{prem}]_{rel}$ such that its (unique) constructive equality in $[C_{concl}]_{constr\_eq}$ is $(t = t')$ where t' is a Skolem term.

Example 1.3.21. *Let C_{prem} = {$r \in R$} and C_{concl} = {$r \in R$, $s \in S$, $s.B = r.A$, $s.C = f(r.A)$}, $[C_{concl}]_{constr\_eq}$ = {$s.B = r.A$, $s.C = f(r.A)$}.*

Then C is an sk_TGD, $DTPrem(C)$ = {$r.A$} and $s.C$ is a Skolem-determined term.

Sk_constraints from regular constraints. Note that while sk_bodies are a generalization of $bodies$, this is not the case for $sk\_constraints$ vs. regular constraints. We hereafter show how to "transform" regular constraints into $sk\_constraints$ by means of their *sk_form*:

Definition 1.3.22 (Sk_form of a constraint)**.** *Let C be a constraint in normalized form. We define the* sk_form *of C as the sk_constraint $sk(C)$=($sk(C)_{prem}$, $sk(C)_{concl}$), where $sk(C)_{prem}$ = C_{prem} and:*

1. *if C is an EGD, then $sk(C)_{concl}$ = C_{concl}*
2. *else, if C is a TGD, let $E_{free} = \{E_1, E_2, \ldots, E_f\}$ be the (possibly empty) subset of the equivalence classes induced by $Clos([C_{concl}]_{eq})$ that contain specific conclusion terms, but do not contain any distinguished premise term or any constant. Let {$t_1, \ldots, t_s$} be the set of distinguished premise terms of C.*

   We associate to each $E_k \in E_{free}$ a Skolem function symbol *of arity s f_k^C. and the* Skolem term $f_k^C(t_1, t_2, \ldots, t_s)$. *Note that the Skolem function symbols thus produced are* distinct among them *and* specific to the constraint C.

   We construct the sk_body $sk(C)_{concl}$ by:

   - *letting $[sk(C)_{concl}]_{rel}$ = $[C_{concl}]_{rel}$, $[sk(C)_{concl}]_{constr\_eq}$= ϕ, $[sk(C)_{concl}]_{eq}$ = ϕ*
   - *adding to $[sk(C)_{concl}]_{constr\_eq}$, all equality atoms in $[C_{concl}]_{eq}$ that contain one distinguished premise term or one constant*
   - *adding to $[sk(C)_{concl}]_{constr\_eq}$, for every $s_i.A \in E_k$, the equality $s_i.A = f_k^C(t_1, t_2, \ldots, t_s)$*

Note that that for a constraint C, the distinguished premise terms of C are also the distinguished premise terms of $sk(C)$.

Example 1.3.23. *Consider the TGD b_{V_R} of Example 1.1.2:*
b_{V_R}: $\forall v_r, v_r \in V_R \rightarrow \exists r, r \in R \wedge r.A = v_r.A \wedge r.C = v_r.C$

Let C be the expression using bodies of b_{V_R}. Then C is such that:

- C_{prem}={$v_r \in V_R$}
- C_{concl}={$v_r \in V_R$, $r \in R$, $r.A = v_r.A$, $r.C = v_r.C$}.

The sk_form of C, $sk(C)$, is then such that:

- $sk(C)_{prem}$ = *{$v_r \in V_R$}*
- $sk(C)_{concl}$ = *{$v_r \in V_R$, $r \in R$, $r.A = v_r.A$, $r.C = v_r.C$, $r.B = f(v_r.A, v_r.C)$}, where all equalities are constructive.*

Note how producing the sk_form of a TGD involves "providing an identity" for all the terms of the conclusion. Note further that the following happens:

1. the equalities in the conclusion of C involving a premise term are *transformed* into constructive equalities (this is the case for the equalities $r.A = v_r.A$ and $r.C = v_r.C$ in the example above)

2. the other equalities are replaced by individual constructive equalities with a Skolem term and the corresponding terms, those that were not equated to premise terms, become therefore Skolem-determined terms in $sk(C)$. All terms that do not participate initially in an equality also become Skolem-determined. This is indeed the case for $r.B$ in the example above.

One may wonder if, given that the original equalities of the Skolem-determined terms are lost, they can be in any way retrieved from the sk_form of C. The answer to this question is clearly yes: they can be retrieved on the *closed version* of the conclusion of $sk(C)$. Indeed, one can show that the above procedure of producing the sk_form of a constraint always ensures that $[\overline{sk(C)_{concl}}]_{eq} = [\overline{C_{concl}}]_{eq}$. In other words, the following holds:

Proposition 1.3.24. *Let C be a constraint. Then $\overline{C_{concl}} = Body(\overline{sk(C)_{concl}})$.*

Note that the above statement not only specifies that the original equalities can be retrieved, but it further states that no "parasite" equalities are introduced in the sk_form among projection terms of the conclusion.

For a set of constraints $\mathcal{C}$, we will denote by $sk(\mathcal{C})$ the set of $sk\_constraints$ $sk(\mathcal{C}) = \{sk(C), C \in \mathcal{C}\}$. We will in following sections show that Standard Chasing with a set of constraints $\mathcal{C}$ and Conservative Chasing with their sk_form $sk(\mathcal{C})$ leads to equivalent results. Not surprisingly, this equivalence will be ensured by Proposition 1.3.24 and by the fact that for each $sk\_constraint$ produced, the Skolem function symbols employed are fresh, thus non-conflicting with other $sk\_constraints$ in $sk(\mathcal{C})$. This will then ensure that no Skolem-determined terms will be "mixed-up" and wrongly equated in the cs_chase results.

1.3.3.2 Homomorphisms of sk_bodies

As is the case of the Standard Chase, the cs_chase also relies on $homomorphisms$. Following the generalization of $bodies$ to sk_bodies, we hereafter show how to extend the notion of $homomorphism$ to sk_bodies:

Definition 1.3.25 (Homomorphisms of sk_bodies)**.** *Let h be a function from the tuple variables of sk_body B_1 into tuple variables of sk_body B_2. Based on h, we can define two additional functions:*

1. *a function h_{terms} over $T(B_1)$, such that:*
   - $h_{terms}(r.A) = h(r).A$, *for $r.A$ a projection term in $T(B_1)$*
   - $h_{terms}(K) = K$, *for K a constant term in $T(B_1)$*
   - $h_{terms}(f(t_1, \ldots, t_n)) = f(h_{terms}(t_1), \ldots, h_{terms}(t_n))$ *for $f(t_1, \ldots, t_n)$ a Skolem term in $T(B_1)$. In particular, $h_{terms}(f()) = f()$ for a Skolem term with no argument.*
2. *a function h_{atoms} over $\overline{B_1}$, such that:*
   - $h_{atoms}(\ (r \in R)\ ) = (h(r) \in R)$, *for $(r \in R)$ in $[B_1]_{rel}$*
   - $h_{atoms}(\ (t_1 = t_2)\ ) = (h_{terms}(t_1) = h_{terms}(t_2))$, *for $(t_1 = t_2)$ in $[\overline{B_1}]_{eq}$*
   - $h_{atoms}(\ (t_1 = t_2)\ ) = (\ h_{terms}(t_1) = ConstrT(h_{terms}(t_2))\ )$, *for $(t_1 = t_2)$ in $[\overline{B_1}]_{constr\_eq}$*

Then h is a homomorphism iff:

1. *for each relational atom a in B_1, $h_{atoms}(a)$ is in $[B_2]_{rel}$.*
2. *for each equality atom a in $[B_1]_{eq}$, the equality atom $h_{atoms}(a)$ is in $[B_2]_{eq}$.*
3. *for each equality atom a in $[B_1]_{constr\_eq}$, the equality atom $h_{atoms}(a)$ is in $[B_2]_{constr\_eq}$.*

As we have done for functions defined on the tuple variables of $bodies$, we will use h to also denote h_{terms} and h_{atoms}. Note that the above definition is indeed a generalization of $homomorphisms$ of $bodies$, where we further impose a restriction on the images of constructive equalities.

Example 1.3.26. *Let B = {$r \in R$, $t \in T$, $s \in S$, $t.D = r.A$, $s.B = r.A$, $s.C = f(r.A)$} be an sk_body, where all equalities are constructive. Note that $[B]_{b-rel}$ = {$r \in R$}*

Let B_1 = {$t_1 \in T$, $s_1 \in S$, $s_1.B = t.D$, $s_1.C = f(t.D)$}.

Then h = {$t_1 \to t$, $s_1 \to s$} is a homomorphism from B_1 to B. Indeed, note that in B, $ConstrT(h(t_1.D))$ = $r.A$.

We can further show that the following holds for $homomorphisms$ of sk_bodies:

Proposition 1.3.27. *Let h be a homomorphism from an sk_body B_1 to an sk_body B_2. Let a be a term in $T(B_1)$. Then $ConstrT(h(ConstrT(a))$ = $ConstrT(h(a))$.*

Proof. Indeed, if a is in $T([B_1]_{b-rel})$ or a is a Skolem term without arguments, then $ConstrT(a)$ = a, and the equality trivially holds.

Else, if a is a projection term, then let $a = ConstrT(a)$ be the unique constructive equality of a. Since h is a $homomorphism$, it follows that $(h(a) = ConstrT(h(ConstrT(a)))$ is

in $[B_2]_{constr\_eq}$. But since B_2 is an sk_body the only constructive equality of $h(a)$ in B_2 is $h(a) = ConstrT(h(a))$. Therefore, $ConstrT(h(ConstrT(a))$ = $ConstrT(h(a))$.

If a is a Skolem term of the form $f(a_1, \ldots, ..a_n)$ then $h(a)$ = $f(h(a_1), \ldots, h(a_n))$. Then $ConstrT(h(a))$ = $f(ConstrT(h(a_1)), \ldots, ConstrT(h(a_n)))$. On the other hand we can further develop $ConstrT(h(ConstrT(a)))$ = $f(ConstrT(h(ConstrT(a_1))), \ldots,$ $ConstrT(h(ConstrT(a_1)))$ (the above developments are all enabled by the fact that constructive terms and Skolem functions commute). By induction on the Skolem terms arguments we can thus prove the required equality. □

Based on Proposition 1.3.27, one can show that the composition of two $homomorphisms$ of sk_bodies is also an $homomorphism$. If h is bijective (on the tuple variables) and h^{-1} is a $homomorphism$ from B_2 to B_1, we call h an *isomorphism*.

Let B_1 and B_2 be two sk_bodies. If there exists an isomorphism between B_1 to B_2 they are said to be isomorphic. If there exists h $homomorphism$ from B_1 to B_2 and h' $homomorphism$ from B_2 to B_1, they are said to *homomorphically equivalent*.

We can further show that the following holds:

Proposition 1.3.28. *Let h be a $homomorphism$ from an sk_body B_1 to an sk_body B_2. Then:*

1. *h is a $homomorphism$ from $\overline{B_1}$ to $\overline{B_2}$*
2. *h is a $homomorphism$ from $Body(B_1)$ to $Body(B_2)$*

Compatibility of $homomorphisms$ of sk_bodies. We extend the notion of compatibility to $homomorphisms$ of sk_bodies. Let h be a $homomorphism$ from an sk_body B_1 to an sk_body B and h' be a $homomorphism$ from an sk_body B_2 to an sk_body B'. We say that h and h' are *compatible* if $h' = h$ on $TupVar(B_1) \cap TupVar(B_2)$.

1.3.3.3 Conservative Chase steps and sequences

We are now ready to formally define the cs_chase steps. A cs_chase step will take as input an sk_body and an $sk\_constraint$ and will yield as output an sk_body. As was the case for the Standard Chase steps, we will present cs_chase steps by first listing their conditions of application and then by describing their application, that is, how they produce an output sk_body given an input sk_body and an $sk\_constraint$.

Definition 1.3.29 (cs_chase step conditions of application)**.** *A cs_chase step with $sk\_constraint$ C on an sk_body B applies iff:*

1. *There exists a $homomorphism$ h from C_{prem} to $\overline{B}$*
2. *There exists no $homomorphism$ h' compatible with h from C_{concl} to $\overline{B}$.*

Definition 1.3.30 (cs_chase step application)**.** *The application of an cs_chase step with $sk\_constraint$ C on an sk_body B, given $homomorphism$ h from C_{prem} to $\overline{B}$, results in a new sk_body B' = $CS\_Chase\_Step\_Res(B, C, h)$ such that $B' \supset B$ and B' is obtained from B as follows:*

1. *let* $B' = B$
2. *add to* B' *the relational atoms* $s'_1 \in S_1, \ldots, s'_n \in S_n$ *(if any), using fresh tuple variables (one for each relational atom specific to* C_{concl}*)*
3. *define the function* h' *from the tuple variables of* C_{concl} *into* B' *such that*
   (a) $h'(r) = h(r)$ *for each tuple variable* r *in* $C_{prem} \cap C_{concl}$
   (b) $h'(s_j) = s'_j$*, for each remaining tuple variable* s_j *in* C_{concl}
4. *for each equality atom* eq *in* $[C_{concl}]_{constr\_eq}$*, add the equality atom* $h'(eq)$ *to* $[B']_{constr\_eq}$
5. *for each equality atom* eq *in* $[C_{concl}]_{eq}$*, add the equality atom* $h'(eq)$ *to* $[B']_{eq}$*.*

As was the case for the Standard Chase, it is easy to show that the function h' constructed in the cs_chase step application on an sk_body B is a $homomorphism$ compatible with h, from C_{concl} to B'. Similar to the case of Standard Chase steps, we will hereafter call h' the cs_chase step compatible $homomorphism$.

Example 1.3.31. *Let* B_1 = *{*$v_r \in V_R$, $v_s \in V_S$, $v_t \in V_T$, $v_r.C$=$v_s.C$, $v_s.D$=$v_t.D$*} be the* sk_body *(which is also a body) corresponding to* R_1 *in Example 1.1.2.*

Let C *be the sk_form of the constraint* b_{V_R}*,* C_{prem} = *{*$v_r \in V_R$*}, and* C_{concl} = *{*$v_r \in V_R$, $r \in R$, $r.A = v_r.A$, $r.C = v_r.C$, $r.B = f(v_r.A, v_r.C)$*}.*

Then a cs_chase *step with* C *applies on* B_1*, yielding* B'_1 = *{*$v_r \in V_R$, $v_s \in V_S$, $v_t \in V_T$, $r \in R$, $v_r.C$=$v_s.C$, $v_s.D$=$v_t.D$, $r.A = v_r.A$, $r.C = v_r.C$, $r.B = f(v_r.A, v_r.C)$*}, where*

1. $[B'_1]_{rel}$ = *{*$v_r \in V_R$, $v_s \in V_S$, $v_t \in V_T$, $r \in R$*}*
2. $[B'_1]_{eq}$ = *{*$v_r.C$=$v_s.C$, $v_s.D$=$v_t.D$*}*
3. $[B'_1]_{constr\_eq}$ = *{*$r.A = v_r.A$, $r.C = v_r.C$, $r.B = f(v_r.A, v_r.C)$*}*

Conservative Chase sequences. Given an sk_body B and a set of $sk\_constraints$ $\mathcal{C}$, a cs_chase sequence consists in producing the sk_bodies B_0, B_1, …, such that:

1. B_0 = B
2. B_i is obtained from B_{i-1} by the following operations:
   (a) pick $C \in \mathcal{C}$ s.t. a cs_chase step with C applies on B_{i-1}, with a $homomorphism$ h from C_{prem} to $\overline{B_{i-1}}$;
   (b) let B_i :=$CS\_Chase\_Step\_Res(B_{i-1}, C, h)$;

For a finite cs_chase sequence with a number of steps k, we denote by the *result* of the sequence the sk_body B_k produced by the last step.

A *full* cs_chase sequence consists in applying cs_chase steps as long as there exists at least an $sk\_constraint$ $C \in \mathcal{C}$ such that a cs_chase with C applies. A *terminating* cs_chase sequence is a full cs_chase sequence that terminates after a finite number of steps n – that is, B_n

is such that for any $sk\_constraint\ C$ in $\mathcal{C}$, and any possible $homomorphism\ h$ from C_{prem} to $\overline{B_n}$, there exists a compatible $homomorphism$ from C_{concl} to $\overline{B_n}$.

Conservative Chase sequences over bodies and queries. As already mentioned, we will be particularly interested in the following in those cs_chase sequences starting from a regular $body$. In particular, we will exhibit strong equivalence results between such sequences and *Standard Chase* sequences over the same $body$.

We cannot however straightforwardly translate intermediate cs_chase steps to corresponding steps on queries. Indeed, the way to infer a query from an sk_body would be to go through the canonical associated body. On the other hand, the transformation from sk_bodies to $bodies$ is not lossless (given an $sk\_body\ B$, $Body(B)$ is in general not equal to B). Then we would lose some of the conditions of application for the next cs_chase step.

However, we can apply such transformation on the *result* of a cs_chase sequence. Given a query Q and a finite cs_chase sequence on $body$(Q) resulting in an $sk\_body\ B'$, we thus define the *result of the* cs_chase *sequence* on Q as the query $Q' = Query(Head(Q), Body(\overline{B'}))$.

1.3.3.4 Properties of terminating Conservative Chase sequences

As is the case for the Standard Chase sequences, full Conservative Chase sequences are not guaranteed to terminate. We will show hereafter that when they *do terminate* however, as was the case of the Standard Chase (Theorem 1.3.12), they lead to equivalent results, as follows:

Theorem 1.3.32. *Let B be an sk_body and $\mathcal{C}$ a set of $sk\_constraints$. Let B_1 and B_2 be the results of two terminating cs_chase sequences with $\mathcal{C}$ over B.*

Then $\overline{B_1}$ and $\overline{B_2}$ are homomorphically equivalent.

To prove the above, we will rely on the fact that, by definition of a cs_chase step, the added image of the conclusion exhibits a "one to one" correspondence with the (specific part of) the conclusion. This particular property allows us to derive $homomorphisms$ over the output of a cs_chase step, based on the existence of $homomorphisms$ on the input of the cs_chase step, as follows:

Lemma 1.3.33. *Let B be an sk_body and C an $sk\_constraint$ such that a cs_chase step with C applies on B with $homomorphism$ h from C_{prem} to $\overline{B}$, yielding B'=$CS\_Chase\_Step\_Res(B, C, h)$.*

Let H be a $homomorphism$ from $\overline{B}$ to an sk_body D. Let $g = H \circ h$ be the corresponding $homomorphism$ from C_{prem} to D.

If there exists a $homomorphism$ g' compatible with g from C_{concl} to D, then there exists a $homomorphism$ from B' to D.

Proof. Let h' be the cs_chase step compatible $homomorphism$.

If C is an sk_EGD, then we will show that H itself is a $homomorphism$ from B' to D. Indeed, for the unique equality $(t_1 = t_2)$ in $B' - B$, $(t_1 = t_2) = h'((t'_1 = t'_2))$, where $(t'_1 = t'_2)$ is the unique equality in C_{concl}. Therefore $t_1 = h'(t'_1)$ and $t_2 = h'(t_2)$. Then $H((t_1 = t_2))$ = $(H(t_1) = H(t_2))$ = $(H \circ h'(t'_1) = H \circ h'(t'_2))$ = $(H \circ h(t'_1) = H \circ h(t'_2))$ = $(g(t'_1) = g(t'_2))$ = $(g'(t'_1) = g'(t'_2))$ = $g'(t'_1 = t'_2)$, therefore $H((t_1 = t_2)) \in D$, where we have used the fact that h' is compatible with h, g' is compatible with g and $t'_1, t'_2 \in T(C_{prem})$.

If C is an sk_TGD, we start by noting that, as mentioned above, the cs_chase compatible $homomorphism$ creates a one-to-one correspondence between the tuple variables of $[C_{concl}]_{rel}$ -$[C_{prem}]_{rel}$ (i.e. the tuple variables specific to the conclusion of C) and the tuple variables of $B' - B$, as well as the corresponding relational atoms. We formalize this observation by stating that there is a partial inverse of h', h'^{-1}, such that the following hold:

1. h'^{-1} is a $homomorphism$ from $[B']_{rel}$-$[B]_{rel}$ to $[C_{concl}]_{rel}$-$[C_{prem}]_{rel}$
2. for every (constructive) equality $(t_1 = t_2)$ in $B' - B$, $(t_1 = t_2) = h'((t'_1 = t'_2))$, where $(t'_1 = t'_2)$ is an equality in $[C_{concl}]_{constr\_eq}$, and furthermore:
   (a) if t_1 is in $T(B')$ - $T(B)$, then $t_1 = h'(t'_1)$, t'_1 is in $T([C_{concl}]_{rel} - [C_{prem}]_{rel})$ and $t'_1 = h'^{-1}(t_1)$
   (b) else, t_2 is in $T(B)$, $t_2 = ConstrT(h'(t'_2))$, $t'_2 \in T(C_{prem})$

Based on the observation above, we define the following function from $TupVar(B')$ to $TupVar(D)$:

$$H'(r) = \begin{cases} H(r), & r \in TupVar(B) \\ g' \circ h'^{-1}(r), & r \in TupVar(B') - TupVar(B) \end{cases}$$

.

We will show that H' is a $homomorphism$ from B' to D. It is straightforward that the image of all the relational atoms in B' is in D. Moreover, all equalities in $B' - B$ are *constructive* and for every equality atom $(t_1 = t_2)$ in $B' - B$, $H'((t_1 = t_2)) = (H'(t_1) = ConstrT(H'(t_2)))$.

But $H'(t_1) = g' \circ h'^{-1}(t_1) = g' \circ h'^{-1} \circ h'(t'_1) = g'(t'_1)$

On the other hand $ConstrT(H'(t_2)) = ConstrT(H(ConstrT(h'(t'_2))))$. But then according to Proposition 1.3.27, $ConstrT(H'(t_2)) = ConstrT(H \circ h'(t'_2)) = ConstrT(H \circ h(t'_2)) = ConstrT(g(t'_2)) = ConstrT(g'(t'_2))$, where we have used the fact that h and h', respectively g and g' are compatible and t'_2 is in $T(C_{prem})$.

It follows that $H'((t_1 = t_2)) = (H'(t_1) = ConstrT(H'(t_2))) = (g'(t'_1) = ConstrT(g'(t'_2)))$, therefore, since g' is a $homomorphism$ from C_{concl} to D, $H'((t_1 = t_2)) \in D$. □

Based on the results above, we are now ready to prove Theorem 1.3.32:

Proof of Theorem 1.3.32. Let $S_0 = B, \ldots, S_n = B_1$ be the terminating cs_chase sequence leading to B_1. We will show by induction on the cs_chase steps the existence of a $homomorphism$ h^1_t from $\overline{S_t}$ to $\overline{B_2}$.

Indeed, since $S_0 = B \subseteq B_2$, $h^1_0 = Id$ is a $homomorphism$ from S_0 to B_2, therefore by Proposition 1.3.28 from $\overline{S_0}$ to $\overline{B_2}$.

Assuming the existence of h^1_t, we will show the existence of h^1_{t+1}.

Indeed, $t \longrightarrow t+1$ is a cs_chase step with an $sk\_constraint$ $C \in \mathcal{C}$. Then there exists a $homomorphism$ h from C_{prem} to $\overline{S_t}$. It follows that $g = h^1_t \circ h$ is a $homomorphism$ from C_{prem} to $\overline{B_2}$. But since B_2 is the result of a terminating cs_chase sequence, it follows that there exists g' a $homomorphism$ compatible with g, from C_{concl} to $\overline{B_2}$. Then we are in the conditions of Lemma 1.3.33, and it follows that there exists h^1_{t+1} a $homomorphism$ from S_{t+1} to $\overline{B_2}$. By Proposition 1.3.28, h^1_{t+1} is then also a $homomorphism$ from $\overline{S_{t+1}}$ to $\overline{B_2}$.

Accordingly, there exists a $homomorphism\ h_n^1$ from $\overline{S_n} = \overline{B_1}$ to $\overline{B_2}$. We show in an identical fashion the existence of a $homomorphism$ from $\overline{B_2}$ to $\overline{B_1}$, thus concluding our proof.

□

1.3.3.5 The Conservative Chase and the Standard Chase

While in the previous subsection we have shown equivalence for terminating cs_chase sequences, we dedicate this section to showing that (as announced in Section 1.2 and restated in previous paragraphs), the Conservative Chase and the Standard Chase lead in essence to equivalent results, as follows:

Theorem 1.3.34. *Let B be a body and $\mathcal{C}$ a set of constraints.*

Let B_1 be the result of a terminating Standard Chase sequence with $\mathcal{C}$ on B. Let B_2 be the result of a terminating cs_chase sequence with $sk(\mathcal{C})$ on B.

Then $\overline{B_1}$ and $Body(\overline{B_2})$ are homomorphically equivalent.

While the above theorem may look cryptic in terms of the equivalence it exhibits, we restate it below, based on Proposition 1.3.7 and the definitions of the corresponding chase flavours on queries:

Corollary 1.3.35. *Let Q be a query and $\mathcal{C}$ a set of constraints.*

Let Q_1 be the result of a terminating Standard Chase sequence with $\mathcal{C}$ on Q. Let Q_2 be the result of a terminating cs_chase sequence with $sk(\mathcal{C})$ on Q.

Then Q_1 and Q_2 are equivalent.

To prove Theorem 1.3.34, we start by showing how the definition of a Standard Chase step allows inferring $homomorphisms$ on the output of the chase step, based on the existence of $homomorphisms$ on the input of the chase step, in a very similar fashion to Lemma 1.3.33:

Lemma 1.3.36. *Let B be a body and C a constraint such that a Standard Chase step with C applies on B with homomorphism h from C_{prem} to $\overline{B}$, yielding $B' = chase\_step\_res(B, C, h)$.*

Let H be a homomorphism from $\overline{B}$ to a body D. Let $g = H \circ h$ be the corresponding homomorphism from C_{prem} to D.

If there exists a homomorphism g' compatible with g from C_{concl} to D, then there exists a homomorphism from B' to D.

Proof. Let h' be the Standard Chase step compatible $homomorphism$.

If C is an EGD, then we will show that H is a $homomorphism$ from B' to D. Indeed, for the unique equality $(t_1 = t_2)$ in $B' - B$, $(t_1 = t_2) = h((t'_1 = t'_2))$, where $(t'_1 = t'_2)$ is the unique equality in C_{concl}. Therefore $t_1 = h(t'_1)$ and $t_2 = h'(t_2)$. Then $H((t_1 = t_2))$ = $(H(t_1) = H(t_2))$ = $(H \circ h'(t'_1) = H \circ h'(t'_2))$ = $(H \circ h(t'_1) = H \circ h(t'_2))$ = $(g(t'_1) = g(t'_2))$ = $(g'(t'_1) = g'(t'_2))$ = $g'((t'_1 = t'_2))$, therefore $H((t_1 = t_2)) \in D$, where we have used the fact that h is compatible with h', g' is compatible with g and $t'_1, t'_2 \in T(C_{prem})$.

If C is a TGD, then we start by noting that the Standard Chase step compatible $homomorphism$ creates a one-to-one correspondence between the tuple variables of $[C_{concl}]_{rel}$ - $[C_{prem}]_{rel}$ and

the tuple variables of $B' - B$, as well as the corresponding relational atoms. We formalize this observation by stating that there is a partial inverse of h', h'^{-1}, such that the following hold:

1. h'^{-1} is a *homomorphism* from $[B']_{rel}$-$[B]_{rel}$ to $[C_{concl}]_{rel}$-$[C_{prem}]_{rel}$
2. for every equality $(t_1 = t_2)$ in $B' - B$, $(t_1 = t_2) = h'((t'_1 = t'_2))$, where $(t'_1 = t'_2)$ is an equality in C_{concl}, and furthermore if $t_i \in T(B') - T(B)$ then $t'_i = h'^{-1}(t_i)$, else $t_i = h'(t'_i)$ and $t'_i \in T(C_{prem})$

Based on the observation above, we define the following function from $TupVar(B')$ to $TupVar(D)$:

$$H'(r) = \begin{cases} H(r), & r \in TupVar(B) \\ g' \circ h'^{-1}(r), & r \in TupVar(B') - TupVar(B) \end{cases}$$

.

We will show that H' is a *homomorphism* from B' to D. It is straightforward that the image of all relational atoms in B' is in D. Moreover, for every equality atom $(t_1 = t_2)$ in $h'(C_{concl})$, $H'((t_1 = t_2)) = (H'(t_1) = H'(t_2))$.

If $t_i \in T(B') - T(B)$, then $H'(t_i) = g' \circ h'^{-1}(t_i) = g' \circ h'^{-1} \circ h'(t'_i) = g'(t'_i)$.

Else, $H'(t_i) = H(t_i) = H \circ h'(t'_i) = H \circ h(t'_i) = g(t'_i) = g'(t'_i)$, where we have used the fact that h' is compatible with h and g' is compatible with g.

Then $H'((t_1 = t_2)) = (g'(t'_1) = g'(t'_2)) = g'((t'_1 = t'_2))$, where $(t'_1 = t'_2) \in [C_{concl}]_{eq}$, so since g' is a *homomorphism* from C_{concl} to D it follows that $H'((t_1 = t_2)) \in D$ □

We will continue by showing that a similar result holds in the case of the cs_chase steps, but concerning the *bodies* recovered from the corresponding (closed versions) of the sk_bodies. To exhibit this result, we first show a set of properties regarding the cs_chase steps and constructive terms.

We start by showing how cs_chase steps with sk_TGDs can be characterized according to the constructive terms. Indeed, as a direct consequence of the definition of the cs_chase steps and $sk\_constraints$, the following holds:

Proposition 1.3.37. *Let B be an sk_body and C an sk_TGD such that a cs_chase step with C applies on B with homomorphism h from C_{prem} to $\overline{B}$, yielding B' = $CS\_Chase\_Step\_Res(B, C, h)$. Let h' be the cs_chase step compatible homomorphism.*

Let $a_1, \ldots, a_n$ be the distinguished premise terms of C.

Let $(t_1 = t_2)$ be a constructive equality in C_{concl}. Then:

1. *if $t_1 = a_i$ then in h'(C_{concl}) the unique constructive equality of $h'(t_1)$ is $(h'(t_1) = ConstrT(h(a_i)))$*
2. *else $t_1 = f_k(a_1, \ldots, a_n)$ and the unique constructive equality of $h'(t_1)$ is $(h'(t_1) = f_k(ConstrT(h(a_1)), \ldots, ConstrT(h(a_n))))$.*

Based on the above, we can show that in a cs_chase sequence starting from an sk_body B with a set of $sk\_constraints$ $\mathcal{C}$, a cs_chase step with an sk_TGD $C \in \mathcal{C}$ *cannot apply twice for the same constructive terms of the images of distinguished premise terms.*

The intuition behind this is simple: since according to Proposition 1.3.37 the constructive terms in the added image of the conclusion are determined by the constructive terms of the images of the distinguished premise terms, the image of the conclusion added by a first cs_chase step with the sk_TGD will provide a compatible $homomorphism$ for the second attempt of a cs_chase step with the same constraint:

Proposition 1.3.38. *Let B be an sk_body and $\mathcal{C}$ a set of $sk\_constraints$.*

Let $B_0 = B, B_1, \ldots, B_n$ be a cs_chase sequence of B with $\mathcal{C}$. For each cs_chase step, let C_i be the corresponding $sk\_constraint$, h_i the $homomorphism$ from $C_{i\,prem}$ to $\overline{B_i}$ and h'_i the cs_chase step compatible $homomorphism$.

Let $C \in \mathcal{C}$ be a sk_TGD such that a cs_chase step with C applies on B_n, with a $homomorphism$ h from C_{prem} to $\overline{B_n}$. Let $a_1, \ldots, a_n$ be the distinguished premise terms of C.

If there exists C_i such that $C = C_i$, then there exists at least one distinguished premise term a_j such that $ConstrT(h(a_j)) \neq ConstrT(h_i(a_j))$

Proof. Assume that this is not the case. Then according to Proposition 1.3.37, the following function:

$$h'(r) = \begin{cases} h(r), & r \in TupVar(C_{prem}) \\ h'_i(r), & r \in TupVar(C_{concl}) - TupVar(C_{prem}) \end{cases}$$

is a $homomorphism$ compatible with h, from C_{concl} to $\overline{B_n}$. Then the cs_chase step with C does not apply. □

Based on Proposition 1.3.38, we will infer a very important result: for a cs_chase sequence starting from a *regular body* with the sk_form of a set of constraints $\mathcal{C}$, the following will hold: for any cs_chase step with an sk_TGD, the constructive terms corresponding to the images of Skolem-determined terms in the conclusion are *new*, that is, they cannot be instantiated in equalities in the input of the chase step.

Intuitively, the reason behind this is that by definition of the $sk\_constraints$, the constructive terms of such Skolem-determined terms will identify uniquely the constructive terms corresponding to images of distinguished premise terms *as well as the $sk\_constraint$ they have been obtained from*, since all $sk\_constraints$ in $sk(\mathcal{C})$ are assumed to use fresh Skolem function symbols. The previous instantiation of such terms would then mean that a cs_chase step with the corresponding constraint has already been applied once, which, as we have previously seen (Proposition 1.3.38), cannot happen.

Proposition 1.3.39. *Let B be an sk_body and $\mathcal{C}$ a set of constraints such that B has been obtained by a cs_chase sequence with $sk(\mathcal{C})$ over a body B_0.*

Let C be an sk_TGD in $sk(\mathcal{C})$ such that a cs_chase step with C applies on B, with a $homomorphism$ h from C_{prem}, yielding $B' = CS\_Chase\_Step\_Res(B_n, C, h)$. Let h' be the cs_chase step compatible $homomorphism$.

Let t be a Skolem-determined term in C_{concl}. Let $t' = h'(t)$ and $(t' = t'')$ be the unique unique constructive equality of t' in B'.

Then there exists no equality involving t'' in B.

Proof. By Proposition 1.3.37, $t'' = f(ConstrT(h(a_1)), \ldots, ConstrT(h(a_n)))$, where $a_1, \ldots, a_n$ are the distinguished premise terms.

Assume there exists a constructive equality in B of the form $v = t''$ where v is a projection term of B. Then since the Skolem function symbol f is *specific to the constraint* C (because of the way $sk(\mathcal{C})$ is obtained, and because B is a $body$, thus contains no initial Skolem terms), it follows that the equality must have resulted from a previous application of a cs_chase step with C.

On the other hand, t'' uniquely determines the constructive terms of the images of the distinguished premise terms. It then follows that there must have been a cs_chase step with C and *the same constructive terms for the images of the distinguished premise terms*. But by Proposition 1.3.38 this cannot happen, thus no equality $v = t''$ can exist in B. □

Note that the idea above has been previously sketched when introducing the sk_form of constraints. Indeed, the above results correspond to the fact that no "parasite" equalities will appear among terms that in the original version of the constraints are *new and specific to the conclusion* (these will be indeed the Skolem-determined terms in the sk_form of the constraints). Such terms can only be equated among them in the Standard Chase, and as shown above, this is equally the case for the cs_chase.

Based on the results exhibited above, we can then state the refinement of Lemma 1.3.33, regarding the $bodies$ corresponding to the output of a cs_chase step, as follows:

Lemma 1.3.40. *Let B be an sk_body and $\mathcal{C}$ a set of constraints, such that B has been obtained from a $body$ B_0 by a cs_chase sequence with $sk(\mathcal{C})$.*

Let $C \in sk(\mathcal{C})$ be an $sk\_constraint$ such that a cs_chase step with C applies on B with $homomorphism$ h from C_{prem} to $\overline{B}$, yielding $B' = CS\_Chase\_Step\_Res(B, C, h)$.

Let H be a $homomorphism$ from $Body(\overline{B})$ to a $body$ D. Let $g = H \circ h$ be the corresponding $homomorphism$ from C_{prem} to D.

If there exists a $homomorphism$ g' compatible with g from $Body(\overline{C_{concl}})$ to D, then there exists a $homomorphism$ from $Body(\overline{B'})$ to $\overline{D}$.

Proof. The proof of the above lemma is very similar to the proof of Lemma 1.3.33 , with the additional usage of Proposition 1.3.39 in the case of sk_TGDs.

We first construct the $body$:

$$B'' = Body(\overline{B}) \cup\ h'(Body(\overline{C_{concl}}))$$

where h' is the cs_chase step compatible $homomorphism$. Using arguments very similar to Lemma 1.3.33, we show that there exists a $homomorphism$ G from B'' to $\overline{D}$ (this $homomorphism$ is either H in the case of an sk_EGD, or H' as defined in the proof of Lemma 1.3.33, using the "invertibility" property of h', in the case of an sk_TGD.)

We will further show that the following holds:

$$Body(\overline{B'}) = \overline{B''}$$

.

By the above and proposition 1.3.28 it follows directly that G is a $homomorphism$ from $Body(\overline{B'})$ to $\overline{D}$.

It is easy to show that $\overline{B''} \subseteq Body(\overline{B'})$, by definition of the closures. The other inclusion is also easy to show in the case of sk_EGDs, since the only added equality in B'' uses projection terms in B.

We will further show the inclusion $Body(\overline{B'}) \subseteq \overline{B''}$ for sk_TGDs. Indeed, according to the definition of the $sk\_constraints$, every equality in $Body(\overline{B'})$ is either in $Body(\overline{B})$ or of the form $(t_1 = t_2)$, where t_1 and t_2 are projection terms and at least one of t_i is in $h'(T(C_{concl}) - T(C_{prem}))$, that is, the image of a projection term specific to the conclusion.

We first analyse the case where t_1 is the image of a non-Skolem-determined term in $T(C_{concl}) - T(C_{prem})$. Assume that t_2 is a projection term of B. Let $(t_1 = t'')$ be the unique equality (constructive) concerning t_1 in B'. Then by definition of the $sk\_constraint$ and the cs_chase step, there exists $t_3 \in h(T(C_{prem}))$ such that $(t_3 = t'')$ is the constructive equality of t_3 in B, and $(t_1 = t_3)$ is in $h'(Body(\overline{C_{concl}}))$, since t_1 and t_3 are projection terms. But then by definition of the closure and since no new equalities are introduced by the cs_chase step on the images of terms of the premise, the equality $(t_3 = t_2)$ must be in $Body(\overline{B})$. It follows that $(t_1 = t_3)$ is in $\overline{Body(\overline{B}) \cup h'(Body(\overline{C_{concl}}))} = \overline{B''}$. The case where t_2 is the image of a non-Skolem determined term in $[C_{concl}]_{rel} - [C_{prem}]_{rel}$ and t_1 is a term of B, as well as the case where both t_1 and t_2 are the images of non-Skolem determined terms specific to the conclusion can be handled in a similar fashion.

We further analyse the case where t_1 is the image of a Skolem-determined term in $T(C_{concl}) - T(C_{prem})$. Let $t_1 = t''$ be the unique constructive equality of t_1 in B'. By Proposition 1.3.39 it follows that the only possible way of equating t_1 and t_2 in $\overline{B'}$ is through t'', and the equality $t_2 = t''$ is in $h'(C_{concl})$. Accordingly, there must exist equalities $(t'_1 = t)$ and $(t'_2 = t)$ in C_{concl}, such that $(t_1 = t'') = h'(t'_1 = t)$ and $(t_2 = t'') = h'(t'_2 = t)$, and t'_1, t'_2 are projection terms specific to the conclusion. Then $(t'_1 = t'_2)$ is in $Body(\overline{C_{concl}})$. It follows $(t_1 = t_2)$ must be in $h'(Body(\overline{C_{concl}}))$, therefore in B'', therefore in $\overline{B''}$, which concludes our proof. □

Remember that when presenting the sk_form of the constraints we have noted that $\overline{C_{concl}} = Body(\overline{sk(C)_{concl}})$. Moreover, recall that $C_{prem} = sk(C)_{prem}$. We can in fact restate the lemma above as follows:

Corollary 1.3.41. *Let B be an sk_body and $\mathcal{C}$ a set of constraints such that B has been obtained from a body B_0 by a cs_chase sequence with $sk(\mathcal{C})$.*

Let $C \in \mathcal{C}$ be a constraint such that a cs_chase step with $sk(C)$ applies on B with $homomorphism$ h from $sk(C)_{prem}$ to $\overline{B}$, yielding $B' = CS\_Chase\_Step\_Res(B, C, h)$.

Let H be a $homomorphism$ from $Body(\overline{B})$ to a body D. Let $g = H \circ h$ be the corresponding $homomorphism$ from C_{prem} to D.

If there exists a $homomorphism$ g' compatible with g from C_{concl} to D, then there exists a $homomorphism$ from $Body(\overline{B'})$ to $\overline{D}$.

Note the strong resemblance of the above corollary with Lemma 1.3.36. Indeed, we are saying that, considering solely the *bodies* of the (closed versions of the) results of cs_chase steps, the cs_chase *behaves like the Standard Chase*. We can then prove our main equivalence result:

Proof of Theorem 1.3.34. Let $S_0 = B$, S_1, ..., $S_n = B_1$ be the terminating Standard Chase sequence with $\mathcal{C}$ on B resulting in B_1.

Let $K_0 = B$, K_1, ..., $K_m = B_2$ be the terminating cs_chase sequence with $sk(\mathcal{C})$ on B resulting in B_2.

A. We will prove by induction on the Standard Chase steps the existence of a $homomorphism$ h_1^t from $\overline{S_t}$ to $Body(\overline{B_2})$.

Since $\overline{S_0} = \overline{B} = Body(\overline{B}) = Body(\overline{K_0}) \subseteq Body(\overline{B_2})$, we can exhibit $h_1^0 = Id$.

Assuming that there exists h_1^t, we will show the existence of h_1^{t+1}, based on Lemma 1.3.36 and Proposition 1.3.24.

Indeed, $t \rightarrow t+1$ is a Standard Chase step with a constraint $C \in \mathcal{C}$. Then there exists a $homomorphism$ h from C_{prem} to $\overline{S_t}$.

Let $g = h_1^t \circ h$. Then g is a $homomorphism$ from C_{prem} to $Body(\overline{B_2})$, therefore a $homomorphism$ from C_{prem} to $\overline{B_2}$. Since B_2 is the result of a terminating cs_chase sequence with $sk(\mathcal{C})$ on B, and C_{prem} = $sk(C)_{prem}$, there must exist a $homomorphism$ g' compatible with g from $sk(C)_{concl}$ to $\overline{B_2}$, therefore (according to Proposition 1.3.28), from $Body(\overline{sk(C)_{concl}})$ to $Body(\overline{B_2})$. By proposition 1.3.24, g' is then a $homomorphism$ compatible with g from $\overline{C_{concl}}$ to $Body(\overline{B_2})$, therefore from C_{concl} to $Body(\overline{B_2})$.

We are then in the conditions of Lemma 1.3.36, and it follows that there exists h_1^{t+1} a $homomorphism$ from S_{t+1} to $Body(\overline{B_2})$, therefore from $\overline{S_{t+1}}$ to $\overline{Body(\overline{B_2})}$ = $Body(\overline{B_2})$.

B. In a very similar manner, we will also prove the existence of a $homomorphism$ h_2^t from $Body(\overline{K_t})$ to $\overline{B_1}$.

Since $Body(\overline{K_0})$= $Body(\overline{B})$ = $\overline{B}$ = $\overline{S_0} \subseteq \overline{B_1}$, we can exhibit $h_2^0 = Id$.

Assuming that there exists h_2^t, we will show the existence of h_2^{t+1}, based on Lemma 1.3.40 and Proposition 1.3.24.

Indeed, $t \rightarrow t+1$ is a cs_chase step with $sk(C)$, where $C \in \mathcal{C}$. Then there exists a $homomorphism$ from $sk(C)_{prem}$ to $\overline{K_t}$, therefore to $Body(\overline{K_t})$ (since $sk(C)_{prem}$ is a regular $body$).

Let $g = h_2^t \circ h$. Then g is a $homomorphism$ from $sk(C)_{prem}$ to $\overline{B_1}$, therefore from C_{prem} to $\overline{B_1}$. Since B_1 is the result of a terminating Standard Chase sequence, it follows that there exists a $homomorphism$ g' compatible with g, from C_{concl} to $\overline{B_1}$, therefore from $\overline{C_{concl}}$ to $\overline{B_1}$. But by Proposition 1.3.24, g' is then a $homomorphism$ compatible with g from $Body(\overline{sk(C)_{concl}})$ to $\overline{B_1}$. We are then in the conditions of Lemma 1.3.40, and it follows there exists h_2^{t+1} a $homomorphism$ from $Body(\overline{K_{t+1}})$ to $\overline{B_1}$, which concludes our proof.

□

1.3.3.6 Termination of the Conservative Chase

We dedicate this subsection to characterizing the termination behaviour of the cs_chase. We start by showing that for a set weakly acyclic constraints $\mathcal{C}$, the termination behaviour of the cs_chase with the sk_form of the constraints, $sk(\mathcal{C})$, is identical to that of the Standard Chase: that is, all sequences terminate within the same type of bounds.

We will further show that cs_chase sequences present a much more *regular* termination behaviour than Standard Chase sequences: that is, we will show that as soon as there exists one terminating cs_chase sequence, *all full* cs_chase *sequences will terminate*.

Weakly acyclic constraints. In the case of weakly acyclic constraints, we present below a result identical to Theorem 1.3.11 in the case of the Standard Chase.

Theorem 1.3.42. *Let B be an sk_body and $\mathcal{C}$ a set of weakly acyclic constraints.*

Then there exists a polynomial in the size of B that bounds the length of every full cs_chase sequence of B with $sk(\mathcal{C})$. In particular, all such sequences terminate.

To prove the above theorem we rely on the following additional result, that merely restates the definition of cs_chase steps conditions of application. Indeed, remember that we have defined a relational atom's identity by means of the constructive terms of all its projection terms, and atoms are considered "identical" if collapsible. The following result underlines the fact that, in order for it to apply, a cs_chase step with an sk_TGD must introduce at least one *new* relational atom (otherwise, there would exist a compatible $homomorphism$ over the conclusion).

Proposition 1.3.43. *Let B be an sk_body and C an sk_TGD such that a cs_chase step with C applies on B, with $homomorphism$ h from C_{prem} to $\overline{B}$, yielding B' =*
$CS\_Chase\_Step\_Res(B, C, h)$.

Then for at least one relational atom a in $B' - B$, there exists no $a' \in [B]_{rel}$ collapsible with a.

Based on the result above and Proposition 1.3.38 (stating that a cs_chase step with an sk_TGD can apply at most once in a sequence), the proof of Theorem 1.3.42 is essentially identical to the proof of Theorem 3.9 in [33], by replacing the notion of "distinct values" with the notion of "distinct constructive terms". Indeed, while the proof in [33] was given for the standard chase, its construction is conservative enough to perfectly account for the cs_chase.

We show below the adaptation of the proof of Theorem 3.9 in [33] to the Conservative Chase. We start by recalling the notion of *weak acyclicity* as defined in [33]:

Definition 1.3.44 (Weakly acyclic set of constraints)**.** *Let $\mathcal{C}$ be a set of constraints over a fixed schema. Construct a directed graph, called the dependency graph of $\mathcal{C}$, as follows:*

1. *there is a node for every pair (R, A), where R is a relation in the schema and A is an attribute of R. Call such pair a* position*.*

2. *add edges as follows: for every TGD C in $\mathcal{C}$:*

   (a) *for every projection term $s_i.A_j$ in $T(C_{concl})$-$T(C_{prem})$ (that is, every projection term specific to the conclusion) and every projection term $r_k.A_m$ in $T(C_{prem})$ such that $s_i.A_j$ and $r_k.A_m$ are in the* same equivalence class *induced by $Clos([C_{concl}]_{eq} \cup [C_{prem}]_{eq})$, add an edge from (R, A_m) to (S, A_j) (if it does not already exist), where $r_i \in R$ and $s_i \in S$ are the relational atoms corresponding to the given projection terms.*

(b) in addition, for every projection term $s_i.A_j$ in $T(C_{concl})$-$T(C_{prem})$ that has no premise term and no constant in its equivalence class, and every projection term $r_k.A_m$ in $T(C_{prem})$ that has a distinguished premise term in its equivalence class, add a special edge *from (R, A_m) to (S, A_j).*

Then $\mathcal{C}$ is said to be weakly acyclic *if there is no cycle going through special edges in its dependency graph.*

Note that the notion of *weak acyclicity* only involves the dependencies among the TGDs in the set of constraints. Note moreover that the original definition of weakly acyclic constraints is expressed for their Datalog notation. The above is the strictly equivalent definition of this concept in the formalism used in this work, the tuple relational calculus.

Finally and importantly, recall that in the sk_form of a set of constraints, all projection attributes specific to the conclusion that are not equated to a constant are either:

- equated to a distinguished premise term. In this case, in the dependency graph there will be *at least* a regular edge from the position corresponding to the distinguished premise term to the conclusion-specific term.
- equated to a Skolem term, that takes as arguments all the distinguished premise terms. In this case, in the dependency graph there will be *at least* special edges from the positions corresponding to the distinguished premise terms to the conclusion-specific term.

The observations above are intended to underline the following: *for a Skolem-determined term in the conclusion of a constraint, in the dependency graph there will always be at least special edges from the positions corresponding to the distinguished premise terms to its corresponding position.*

Example 1.3.45. *Consider the schema $R(A), S(B, C), T(D, E)$ and the sk_body B = {$r \in R$, $s \in S$, $r.A = s.B$}, which is also a regular body, that is, it has no constructive equalities.*

Now consider the set of constraints $\mathcal{C}$ consisting in the unique TGD C such that C_{prem}={$r \in R$, $s \in S$, $r.A = s.B$} and C_{concl} ={$r \in R$, $s \in S$, $t \in T$, $t.D = s.B$}. Note that there is a unique distinguished premise term of C, namely $s.B$.

Then the dependency graph of $\mathcal{C}$ will comprise:

- *two regular edges, one from (R, A) to (T, D) and one from (S, B) to (T, D).*
- *two special edges, one from (R, A) to (T, E) and one from (S, B) to (T, E).*

On the other hand, the sk_form of C is such that $sk(C)_{prem}$ = C_{prem} and $sk(C)_{concl}$ = {$r \in R$, $s \in S$, $t \in T$, $t.D = s.B$, $t.E = f(s.B)$}. Note that equalities in $sk(C)_{concl}$ are all constructive.

By examining these two constructive equalities, note how the dependency graph comprises an edge from the position (S, B) corresponding to $s.B$ to the position (T, D) corresponding to $t.D$. It comprises as well a special edge from the position (S, B) corresponding to $s.B$ to the position (T, E) corresponding to $t.E$.

For conciseness, we will in the following denote by *the set of distinct constructive terms of a position* (R, A) *in an* sk_body B the set of all distinct constructive terms of the projection terms $r_i.A$, where $(r_i \in R)$ is a relational atom in B.

Based on the intuitions above, we can informally sketch the flow of our proof adaptation: we will show that, in the result of any cs_chase sequence with $sk(\mathcal{C})$, where $\mathcal{C}$ is a set of weakly acyclic constraints, there is a bounded number of distinct constructive terms for a given position (R, A) in the resulting sk_body B'. Combined with Proposition 1.3.43, which states that a cs_chase step introduces at least some *fresh* relational atom (that is, one that differs on at least one constructive term from the other atoms corresponding to the same relation), this will then ensure the required bound on the cs_chase steps.

We will rely in our proof, as in [33], on the operations of *copying* and *creation* as they are shown by the dependency graph. The *value copy* (expressed by regular edges) in [33] is in our case replaced by *constructive term copy*, from the image of a distinguished premise term to the added conclusion term. The *value creation* on the other hand, expressed by *special edges*, involves the addition in the cs_chase step of a Skolem determined term, whose constructive term is in turn, as shown above, *completely determined by the Skolem function symbol and the constructive terms of the images of the distinguished premise terms*.

Proof of Theorem 1.3.42. As in [33], we start by analysing the case without EGDs.

In an identical fashion to [33], for every node (R, A) in the dependency graph of $\mathcal{C}$, we define its rank as the maximum number of special edges of any path in the graph ending in (R,A). Since $\mathcal{C}$ is weakly acyclic, the rank of every node will be finite. As in [33], we denote by r the maximum of such ranks, and by p the number of positions in the schema. Since the schema is fixed, we can consider p a constant and we can show that r cannot be higher than p (otherwise a cycle on the special edges will exist), thus r is bounded by a constant.

We then partition, as in [33], the nodes of the dependency graph into sets $N_0, N_1, ..., N_r$, where the set N_i contains all nodes of rank i.

Let n be the total number of distinct constructive terms of projection terms in B. Let B' be an sk_body obtained from B after some arbitrary cs_chase sequence.

We will prove by induction that for every i there exists a polynomial Q_i such that the total number of distinct constructive terms of all positions (R, A) in N_i is bounded by $Q_i(n)$.

If (R, A) is a position in N_0, then there are no incoming paths with special edges. Then *no new constructive terms will be created for a term* $r_i.A$ *corresponding to the position*. Indeed, recall that, since they don't have any incoming special edges, these terms cannot be Skolem-determined. When they are added, their constructive terms will then be *copies of the constructive terms of the image of the distinguished premise term they are equated with in the conclusion*.

Then, for this sort of positions in B', the number of their distinct constructive terms will be at maximum n, corresponding to the initial distinct constructive terms in B.

Assuming that the induction hypothesis holds for a given i, we will show that it also holds for $i + 1$, by analysing the constructive terms for a position (R, A).

The first type of such constructive terms corresponds to constructive terms that already exist in B, thus they are at most n.

Furthermore, a constructive term corresponding to a position (R,A), thus to a projection term $r.A$ where $(r \in R)$ is the relational atom comprising r, can be created in two ways: as a copy

of some previous constructive term (when applying a cs_chase step with a constraint in which $r.A$ is not Skolem-determined) or as a new constructive term, if $r.A$ is Skolem-determined in the conclusion. This new constructive term is then by definition a Skolem term taking as arguments the constructive terms of the images of the distinguished premise terms.

Let us first count the number of *new* distinct constructive terms that can be created for a given position (R, A). A new constructive term creation corresponds to the presence of (at least) an incoming special edge. Therefore, the special edge(s) must originate in some position(s) (S, B) in $N_0 \cup ... \cup N_i$. But according to the induction hypothesis, the number of distinct constructive terms for the positions in $N_0 \cup ... \cup N_i$ is bounded by $P(n) = Q_0(n) + .. + Q_i(n)$.

Let C be a TGD in $\mathcal{C}$ and d_j be the number of its distinguished premise terms. Note that these are the same as the distinguished premise terms of $sk(C)$. We can show that for every distinct choice of d_j constructive terms in the positions of $N_0 \cup ... \cup N_i$, a cs_chase step with $sk(C)$ creates at most one new constructive term for the position (R,A) corresponding to a Skolem-determined term. Indeed, by Proposition 1.3.38, for a given choice of constructive terms for the images of the distinguished premise terms, a constraint will apply at maximum once.

Let d be the maximum number of special edges that may enter a position in the whole dependency graph.

As shown above, by definition of the dependency graph, for each position, d_j is lower or equal than the total number of incoming special edges for a position. Then obviously d_j <= d. Thus the total number of new distinct constructive terms that can be created for a position (R,A) is at maximum $(P(n))^d * D$, where D is the number of TGDs. Since the schema and the number of constraints are assumed to be fixed, the above is a polynomial in n. For the total number of positions (R,A) in N_{i+1}, the number of new distinct constructive terms that can be created is then bounded by $G(n) = p_i * (P(n))^d * D$, where p_{i+1} is the number of positions in N_{i+1}.

Let us now count the number of distinct constructive terms that can be occur for positions in N_{i+1} by *copying*. Such copying may only happen from a position in $N_0 \cup .. \cup N_i$ by the presence of a non-special edge (a copy from a position with a higher rank would contradict the hypothesis that the rank of a position in N_{i+1} is indeed $i + 1$). Thus, the number of distinct constructive terms obtained by copying for positions in N_{i+1} is bounded by the number of distinct constructive terms for positions in $N_0 \cup ... \cup N_i$, which is P_n.

We can then (in an identical fashion as in [33]) take $Q_{i+1}(n) = n + G(n) + P(n)$, the polynomial that bounds the number of distinct constructive terms for the positions in N_{i+1}.

Since the number of sets N_i is bounded by a constant, it follows that the total number of distinct constructive terms for all positions in B' is bounded by a polynomial $Q(n)$, and therefore obviously the number of distinct constructive terms for a given position in B' is itself bounded by $Q(n)$.

It follows that, for a given relation R in the schema, the number of relational atoms $r_i \in R$ differing by at least one constructive term on at least one of their projection terms is bounded by $Q(n)^p$, where p is the number of positions in the schema and therefore an upper bound for the number of attributes of R.

To conclude our proof we note that, by Proposition 1.3.43, each cs_chase step with an sk_TGD introduces *at least one relational atom that is non-collapsible with existing atoms*. It follows that the maximum number of cs_chase steps with sk_TGDs is bounded by $s * Q(n)^p$,

where s is the number of relations in the schema. Since s and p are assumed to be constants (fixed schema), it follows that the total number of cs_chase steps is bounded by a polynomial in n.

Accordingly, we then infer that the number of relational atoms in some resulting sk_body is always bounded by $c * s * Q(n)^p$, where c is the maximum number of atoms in the conclusion of a constraint. To further account for sk_EGDs we note that an sk_EGD will simply equate two projection terms of existing relational atoms. Since the number of such relational atoms is always bounded, it follows that the quantity $c * s * Q(n)^p * p^2$ provides an upper bound for the number of cs_chase steps with EGDs. □

Note that, for the Conservative Chase, the number of special edges on a path between two positions can be in fact related to the *nesting depth* of the Skolem constructive terms (one additional Skolem function symbol is added with each "new constructive term creation", which corresponds to at least one incoming special edge).

While the above statement and proof show the termination of the cs_chase under weakly acyclic constraints, note that the choice of tuple calculus allows for a possibly finer granularity definition of the dependency graph. Indeed, the equalities stated explicitly in the premise in our formalism can help distinguish cases where the standard chase *in tuple calculus* can be shown to terminate beyond weak acyclicity, by the same reasoning as above, but by modifying the definition of the dependency graph. The following example illustrates such a case:

Example 1.3.46. *Consider the schema $R(A), S(B, C), T(D, E)$, the sk_body(which is also a regular body) B = {$r \in R$, $s \in S$, $r.A = s.B$} and the set of constraints $\mathcal{C}$= {C_1, C_2}, such that:*

1. $C_{1\,prem}$ = {$r \in R$, $s \in S$, $r.A = s.B$}, $C_{1\,concl}$ = {$r \in R$, $s \in S$, $r.A = s.B$, $t \in T$, $t.C = s.B$}
2. $C_{2\,prem}$ = {$t \in T$}, $C_{2\,concl}$ ={$t \in T$, $r \in R$, $r.A = t.D$}

Then, by definition, $\mathcal{C}$ is not weakly acyclic. *On the other hand, by applying the same reasoning as above, with a* modified version of the dependency graph*, we can show that the standard chase and the cs_chase terminate. Indeed,* by employing a similar procedure as the one used to produce the sk_form of the constraints*, this modified dependency graph will only comprise* special edges *from* distinguished premise terms *to undetermined conclusion terms (Skolem-determined in the sk_form), that is, in our case, from the position (S, B) to the position (T, E).*

In this alternative dependency graph there are no cycles going through special edges*, thus the reasoning of the proof for weakly acyclic constraints applies directly and will accordingly* infer chase termination*, for both the Standard and Conservative Chase.*

We thus note, with the above example, the possibility of a finer analysis of chase termination conditions, based on the suggested alternative definition of the dependency graph. We leave such refined analysis to future work.

Stronger termination criteria. While in the above we have shown that for weakly acyclic constraints, the cs_chase behaves in an essentially identical fashion to the Standard Chase, we

will hereafter show that the termination of full cs_chase sequences is intuitively much more *regular* than that of full Standard Chase sequences. Mainly, we will show that given a set of $sk\_constraints$, if one full cs_chase sequence terminates then *all full* cs_chase *sequences terminate*.

The claim of such property is based again on the notion of collapsible atoms and atom identity. To formalize our results, we further introduce the notion of $col\_homomorphism$:

Definition 1.3.47 ($col\_homomorphism$)**.** . *We denote by a* $col\_homomorphism$ *from an* sk_body B_1 *to an* sk_body B_2 *a* $homomorphism$ h *from* B_1 *to* B_2 *such that for every relational atom* a *in* B_1, a *and* $h(a)$ *are collapsible.*

Remember that we have seen, by Lemma 1.3.33, that we can infer $homomorphisms$ over the output of a cs_chase step, based on the existence of of $homomorphisms$ over the input of the step. We hereafter refine Lemma 1.3.33 to further include $col\_homomorphisms$ and show the following:

Lemma 1.3.48. *Let* B *be an* sk_body *and* C *an* $sk\_constraint$ *such that a* cs_chase *step with* C *applies on* B *with* $homomorphism$ h *from* C_{prem} *to* $\overline{B}$*, yielding* B'=$CS\_Chase\_Step\_Res(B, C, h)$. *Let* h' *be the* cs_chase *step compatible* $homomorphism$.

Let H *be a* $homomorphism$ *from* $\overline{B}$ *to an* sk_body D. *Let* $g = H \circ h$ *be the corresponding* $homomorphism$ *from* C_{prem} *to* D.

If there exists a $homomorphism$ g' *compatible with* g *from* C_{concl} *to* D, *then there exists a* $homomorphism$ H'' *from* B' *to* D, ***such that, moreover, if*** H ***is a*** $col\_homomorphism$ ***then*** H'' ***is a*** $col\_homomorphism$.

Proof. We only need to show that if H is a $col\_homomorphism$, then the $homomorphism$ H' defined in the proof of Lemma 1.3.33 is also a $col\_homomorphism$. Indeed, for the constructive equalities $(t_1 = t_2)$ in $h'(C_{concl})$, we have shown that $H'((t_1 = t_2)) = (H'(t_1) = ConstrT(H(t_2)))$. Since H is a $col\_homomorphism$ it follows that $ConstrT(H(t_2))$ = $ConstrT(t_2)$, therefore $H'((t_1 = t_2)) = (H'(t_1) = ConstrT(t_2))$, which makes H' a $col\_homomorphism$, thus concluding our proof. □

Using Lemma 1.3.48 and Proposition 1.3.43, we can infer the following very strong result regarding the termination of the cs_chase sequences:

Theorem 1.3.49. *Let* B *be an* sk_body *and* $\mathcal{C}$ *a set of* $sk\_constraints$.

If one full cs_chase *sequence with* $\mathcal{C}$ *over* B *terminates, then all full* cs_chase *sequences with* $\mathcal{C}$ *over* B *terminate.*

Proof. Let B_1 be the result of a terminating cs_chase sequence with $\mathcal{C}$ over B.

Let $S_0 = B, \ldots$ be a full cs_chase sequence with $\mathcal{C}$ over B.

We will show by induction on the cs_chase steps that there exists a $col\_homomorphism$ from $\overline{S_i}$ to $\overline{B_1}$. The reasoning is identical to that of the proof of Theorem 1.3.32, starting from the identity function which is a $col\_homomorphism$ from $S_0 = B$ to B_1, and using Lemma 1.3.48 at each step.

On the other hand, by Proposition 1.3.43, every cs_chase step with an sk_TGD must add at least one *new* relational atom. Since for every such atom there exists a collapsible atom in $\overline{B_1}$,

and B_1 has a finite number of relational atoms, it follows that there exists k such that starting from k all cs_chase steps in the sequence $S_k, \ldots$ are sk_EGD steps. But since S_k has a finite number of relational atoms itself, and every cs_chase step with an sk_EGD adds an equality over existing projection terms, it follows that the number of sk_EGD steps is bounded, thus there exists k_1 such that no more cs_chase step applies on S_{k_1}. Therefore, the $S_0 = B, \ldots$ cs_chase sequence terminates after k_1 steps. □

To conclude, we note that we can refine Theorem 1.3.32 to the following:

Theorem 1.3.50. *Let B be an sk_body and $\mathcal{C}$ a set of $sk\_constraints$. Let B_1 and B_2 be the results of two terminating cs_chase sequences with $\mathcal{C}$ over B.*

Then $\overline{B_1}$ and $\overline{B_2}$ are $col\_homomorphically\ equivalent$.

1.3.3.7 Splitting sk_constraints into sk_unit_constraints

We will in the following further distinguish a subclass of $sk\_constraints$ which we will call $sk\_unit\_constraints$:

Definition 1.3.51 (Sk_unit_constraints)**.** *An $sk\_constraint$ C is an $sk\_unit\_constraint$ iff:*

1. *C is an sk_EGD, or*
2. *C is a sk_TGD and the set $[C_{concl}]_{rel} - [C_{prem}]_{rel}$ contains a single relational atom.*

Intuitively, $sk\_unit\_constraints$ have a *unit* conclusion, in the sense that, ignoring constructive equalities, this conclusion comprises a *single specific atom*. We will show hereafter that cs_chase sequences with $sk\_unit\_constraints$ exhibit less redundancy and even stronger equivalence results upon termination. More importantly, we show that we can "transform" any set of $sk\_constraints$ into a set of $sk\_unit\_constraints$ and the cs_chase with the two versions exhibits essentially similar properties.

We obtain $sk\_unit\_constraints$ from $sk\_constraints$ by producing their *split* form, as follows:

Definition 1.3.52 (Split form of an $sk\_constraint$)**.** *Let C be an $sk\_constraint$. The* split form *of C is a set of $sk\_unit\_constraints$ $split(C)$ obtained as follows:*

1. *if C is an sk_EGD, then $split(C)$ contains an unique element $C_1 = C$*
2. *else, let $r_1 \in R_1, \ldots, r_n \in R_n$ be the relational atoms in C_{concl}. Then $split(C)$ contains n $sk\_unit\_constraints$ C_i, which are all sk_TGDs, constructed as follows:*
   - *(a) $C_{i\,prem} = C_{prem}$*
   - *(b) $[C_{i\,concl}]_{rel} = [C_{prem}]_{rel} \cup (r_i \in R_i)$*
   - *(c) $[C_{i\,concl}]_{constr\_eq} = \{(t_1 = t_2)\}$, s.t. $(t_1 = t_2) \in [C_{concl}]_{constr\_eq}$ and $t_1 = r_i.A$*

Example 1.3.53. *Let C be an sk_TGD such that:*

1. C_{prem} = $\{r \in R\}$
2. C_{concl} = $\{r \in R, s \in S, t \in T, s.B = r.A, s.C = f(r.A), t.D = r.A\}$.

Then the split form of C, $split(C)$, contains two $sk\_unit\_constraints$ C_1 and C_2, both sk_TGDs, such that:

1. $C_{1\,prem}$ = $C_{2\,prem}$ = $\{r \in R\}$
2. $C_{1\,concl}$ = $\{r \in R, s \in S, s.B = r.A, s.C = f(r.A)\}$
3. $C_{2\,concl}$ = $\{r \in R, t \in T, t.D = r.A\}$

For a set of $sk\_constraints$ $\mathcal{C}$, we denote by $split(\mathcal{C})$ the resulting set of $sk\_unit\_constraints$ corresponding to their split versions, $split(\mathcal{C}) = \cup split(C), C \in \mathcal{C}$. As announced, we will in the following show that cs_chase sequences with $\mathcal{C}$ and cs_chase sequences with $split(\mathcal{C})$ behave in an essentially equivalent fashion.

We start by showing that the results of terminating cs_chase sequences with the two versions of constraints are strongly equivalent (that is, $col\_homomorphically\ equivalent$) as follows:

Theorem 1.3.54. *Let B be an sk_body and $\mathcal{C}$ a set of $sk\_constraints$.*

Let B_1 be the result of a terminating cs_chase sequence with $\mathcal{C}$ over B. Let B_2 be the result of a terminating cs_chase sequence with $split(\mathcal{C})$ over B.

Then $\overline{B_1}$ and $\overline{B_2}$ are $col\_homomorphically\ equivalent$.

Proof. We only need to note the fact that, by definition of the split form of the constraints:

1. for a constraint $C \in \mathcal{C}$, a $homomorphism$ h' from C_{concl} to an sk_body D, compatible to a $homomorphism$ h from C_{prem} to D, provides $h'_1, \ldots h'_n$ $homomorphisms$ compatible with h from $C_{i\,concl}$ to D, where $C_i \in split(C)$.
2. reversely, if $h'_1, \ldots h'_n$ exist over $C_{i\,concl}$, $C_i \in split(C)$, then their union will form a $homomorphism$ h' from C_{concl} to D.

We can then exhibit, based on Lemma 1.3.48, a $col\_homomorphism$ to $\overline{B_1}$ from (the closed version of) every intermediate result of the cs_chase sequence with $split(\mathcal{C})$; reversely, we can exhibit a $col\_homomorphism$ from (the closed version of) every intermediate result of the cs_chase sequence with $\mathcal{C}$ to $\overline{B_2}$. □

We further show that the termination behaviour of cs_chase sequences with $\mathcal{C}$ and cs_chase sequences with $split(\mathcal{C})$ is essentially identical, as follows:

Theorem 1.3.55. *Let B be a body and $\mathcal{C}$ a set of $sk\_constraints$.*

Then the following hold:

1. *if there exists one terminating cs_chase sequence with $\mathcal{C}$ over B then all cs_chase sequences with $\mathcal{C}$ and $split(\mathcal{C})$ terminate.*
2. *if there exists one terminating cs_chase sequence with $split(\mathcal{C})$ over B then all cs_chase sequences with $\mathcal{C}$ and $split(\mathcal{C})$ terminate.*

Proof sketch. We proceed in the same fashion as we have for linking termination of cs_chase sequences, based on the $col\_homomorphisms$ exhibited from each intermediate cs_chase result in a sequence to the result of a second, terminating sequence. We have shown in the above proof that such $col\_homomorphisms$ exist in both directions (i.e. from intermediate results of a sequence using $\mathcal{C}$ to a terminating sequence using $split(\mathcal{C})$, and reversely). The reasoning for inferring the termination is then identical to the proof of Theorem 1.3.49. □

Sk_unit_constraints from regular constraints. Given a set of constraints $\mathcal{C}$, we define their sk_unit_form as the set of $sk\_unit\_constraints$ $skunit(\mathcal{C})$ = $split(sk(\mathcal{C}))$.

We can further show that the following holds (the proof is identical to the proof of Theorem 1.3.42):

Theorem 1.3.56. *Let B be an sk_body and $\mathcal{C}$ a set of weakly acyclic constraints:*

Then there exists a polynomial in the size of B that bounds the length of every full cs_chase sequence of B with $skunit(\mathcal{C})$. In particular, all such sequences terminate.

Furthermore, based on Theorem 1.3.34, Theorem 1.3.54 and Proposition 1.3.28, we can claim the following:

Theorem 1.3.57. *Let B be a $body$ and $\mathcal{C}$ a set of constraints.*

Let B_1 be the result of a terminating Standard Chase sequence with $\mathcal{C}$ on B. Let B_2 be the result of a terminating cs_chase sequence with $skunit(\mathcal{C})$ on B.

Then $\overline{B_1}$ and $Body(\overline{B_2})$ are homomorphically equivalent.

Accordingly, we can "translate" the above result for queries:

Corollary 1.3.58. *Let Q be a query and $\mathcal{C}$ a set of constraints.*

Let Q_1 be the result of a terminating Standard Chase sequence with $\mathcal{C}$ on Q. Let Q_2 be the result of a terminating cs_chase sequence with $skunit(\mathcal{C})$ on Q.

Then Q_1 and Q_2 are equivalent.

Privileging $sk\_unit\_constraints$. Given the above equivalence results, we will privilege in the following the $sk\_unit\_constraints$ to express the cs_chase, and its provenance-aware version. Indeed, reasoning in terms of unit conclusions turns out to be simpler. Furthermore, the advantage of employing the cs_chase with the split version of a set of $sk\_constraints$ is that it produces *shorter* outputs, in the following sense:

Proposition 1.3.59. *Let B be a $body$. Let $\mathcal{C}$ be a set of $sk\_unit\_constraints$.*

Then the result of any cs_chase sequence with $\mathcal{C}$ over B does not contain any collapsible atoms.

Moreover, as announced in the beginning of this section, for the results of two terminating cs_chase sequences with a set of $sk\_unit\_constraints$, we can show an equivalence result stronger than for regular $sk\_constraints$. Indeed, we define the notion of $col\_isomorphism$ as follows:

Definition 1.3.60 ($col\_isomorphism$)**.** . *We denote by a $col\_isomorphism$ from an sk_body B_1 to an sk_body B_2 an isomorphism h from B_1 to B_2 such that h is furthermore a $col\_homomorphism$.*

Note that it is straightforward to show that if h is a $col\_isomorphism$ then h^{-1} is also a $col\_isomorphism$.

We can then show that for $sk\_unit\_constraints$ the following stronger version of Lemma 1.3.48 holds:

Lemma 1.3.61. *Let B be an sk_body and C an $sk\_unit\_constraint$ such that a cs_chase step with C applies on B with $homomorphism$ h from C_{prem} to $\overline{B}$, yielding B' = $CS\_Chase\_Step\_Res(B, C, h)$.*

Let H be an isomorphism from $\overline{B}$ to a part P of an sk_body D. Let $g = H \circ h$ be the corresponding $homomorphism$ from C_{prem} to D.

If there exists a $homomorphism$ g' compatible with g from C_{concl} to D, then there exists an isomorphism H'' from B' to $P \cup g'(C_{concl})$ such that, moreover, if H is a $col\_isomorphism$ then H'' is a $col\_isomorphism$.

Proof. The proof of the above result is very similar to the proof of Lemma 1.3.33 and its refinement Lemma 1.3.48.

Let $P' = P \cup g'(C_{concl})$. Let h' be the cs_chase step compatible $homomorphism$.

If C is an sk_EGD, then we will show, as in the proof of Lemma 1.3.33, that H itself is an $homomorphism$ from B' to P'. Indeed, for the unique equality $(t_1 = t_2)$ in $B' - B$, $(t_1 = t_2) = h'((t'_1 = t'_2))$, where $(t'_1 = t'_2)$ is the unique equality in C_{concl}. Therefore $t_1 = h'(t'_1)$ and $t_2 = h'(t_2)$. Then $H((t_1 = t_2))$ = $(H(t_1) = H(t_2))$ = $(H \circ h'(t'_1) = H \circ h'(t'_2))$ = $(H \circ h(t'_1) = H \circ h(t'_2))$ = $(g(t'_1) = g(t'_2))$ = $(g'(t'_1) = g'(t'_2))$ = $g'(t'_1 = t'_2)$, therefore $H(t_1 = t_2) \in P'$, where we have used the fact that h' is compatible with h, g' is compatible with g and $t'_1, t'_2 \in T(C_{prem})$.

On the other hand, we will also prove that H^{-1} is a $homomorphism$ from P' to B'. Indeed, if $g'(C_{concl})$ is not in P, then for the unique equality $(t''_1 = t''_2)$ in $P' - P$, $(t''_1 = t''_2)$ = $g'((t'_1 = t'_2))$. Then $H^{-1}((t''_1 = t''_2)) = (H^{-1}(t''_1) = H^{-1}(t''_2))$ = $(H^{-1} \circ g'(t'_1) = H^{-1} \circ g'(t'_2))$ = $(H^{-1} \circ g(t'_1) = H^{-1} \circ g(t'_2))$ = $(H^{-1} \circ H \circ h(t'_1) = H^{-1} \circ H \circ h(t'_2))$ = $(h(t'_1) = h(t'_2))$ = $h'((t'_1 = t'_2))$ = $(t_1 = t_2)$, which is the unique equality in $B' - B$, thus concluding our proof.

If C is an sk_TGD, we restate the "invertibility" property of h' as in the proof of Lemma 1.3.33: there exists a partial inverse of h', h'^{-1}, such that the following hold:

1. h'^{-1} is a $homomorphism$ from $[B']_{rel}$-$[B]_{rel}$ to $[C_{concl}]_{rel}$-$[C_{prem}]_{rel}$
2. for every (constructive) equality $(t_1 = t_2)$ in $B' - B$, $(t_1 = t_2) = h'(t'_1 = t'_2)$, where $(t'_1 = t'_2)$ is an equality in $[C_{concl}]_{constr\_eq}$, and furthermore:
   (a) if t_1 is in $T(B')$ - $T(B)$, then $t_1 = h'(t'_1)$, t'_1 is in $T([C_{concl}]_{rel} - [C_{prem}]_{rel})$ and $t'_1 = h'^{-1}(t_1)$
   (b) else, t_2 is in $T(B)$, $t_2 = ConstrT(h'(t'_2))$, $t'_2 \in T(C_{prem})$

We then proceed as in the proof Lemma 1.3.33, that is, we define the following function from $TupVar(B')$ to $TupVar(P')$:

$$H'(r) = \begin{cases} H(r), & r \in TupVar(B) \\ g' \circ h'^{-1}(r), & r \in TupVar(B') - TupVar(B) \end{cases}$$

.

We will show that H' is a $homomorphism$ from B' to P', as in the proof of Lemma 1.3.33. It is straightforward that the image of all the relational atoms in B' is in P'. Moreover, all equalities in $B' - B$ are *constructive* and for every equality atom $(t_1 = t_2)$ in $B' - B$, $H'((t_1 = t_2)) = (H'(t_1) = ConstrT(H'(t_2)))$.

But $H'(t_1) = g' \circ h^{-1}(t_1)$ = $g' \circ h'^{-1} \circ h'(t'_1) = g'(t'_1)$

On the other hand $ConstrT(H'(t_2))$ = $ConstrT(H(ConstrT(h'(t'_2))))$. But then according to proposition 1.3.27, $ConstrT(H'(t_2))$ = $ConstrT(H \circ h'(t'_2))$ = $ConstrT(H \circ h(t'_2))$ = $ConstrT(g(t'_2))$ = $ConstrT(g'(t'_2))$, where we have used the fact that h and h', respectively g and g' are compatible and t'_2 is in $T(C_{prem})$.

It follows that $H'((t_1 = t_2)) = (H'(t_1) = ConstrT(H'(t_2)))$ = $(g'(t\_1) = ConstrT(g'(t'_2)))$, therefore , since g' is a $homomorphism$ from C_{concl} to P', $H'((t_1 = t_2)) \in P'$.

In addition to Lemma 1.3.33, we further prove that H' is in fact an isomorphism. That is, we will show that H' is bijective and that its inverse H'^{-1} is a $homomorphism$ from P' to B'.

We start by noting that $g'(C_{concl})$ cannot be in P. Indeed, otherwise it would be easy to show that the following function:

$$h''(r) = \begin{cases} h(r), & r \in TupVar(C_{prem}) \\ H^{-1} \circ g'(r), & r \in TupVar(C_{concl}) - TupVar(C_{prem}) \end{cases}$$

.

would be a $homomorphism$ compatible with h from C_{concl} to $\overline{B}$ *before the* cs_chase *step*, thus the cs_chase step would not apply.

On the other hand, *since we are dealing with* $sk\_unit\_constraints$, it follows by the above that $P \cap g'(C_{concl}) = g([C_{prem}]_{rel})$, in other words, the image of specific part of the conclusion is disjoint from P (indeed, the single relational atom specific to C_{concl} must be outside P).

We further note that in the case of $sk\_unit\_constraints$, *every* compatible $homomorphism$ possesses the "invertibility" property stated above for the cs_chase step compatible $homomorphism$. In particular, g' will exhibit such property.

Then it is easy to show that H' is *bijective*, and for the unique relational atom a in $P' - P$, $H'^{-1}(a)$ = $h' \circ g'^{-1}(a)$ = $h'(a')$, where a' is the unique relational atom in $[C_{concl}]_{rel} - [C_{prem}]_{rel}$, therefore $H'^{-1}(a) \in B'$.

Furthermore, let $(t''_1 = t''_2)$ be a constructive equality in $P' - P$. Then $(t''_1 = t''_2)$ = $g'(t'_1 = t'_2)$. Then $H'^{-1}((t''_1 = t''_2))$ = $H'^{-1} \circ g'(t'_1 = t'_2)$ = $(H'^{-1} \circ g'(t'_1) = ConstrT(H'^{-1} \circ g'(t'_2))$, where $(t'_1 = t'_2)$ is a constructive equality in C_{concl}. But further $H'^{-1} \circ g'(t'_1)$ = $h' \circ g'^{-1} \circ g'(t'_1)$ = $h'(t'_1)$.

Moreover, $ConstrT(H'^{-1} \circ g'(t'_2))$ = $ConstrT(H^{-1} \circ g(t'_2))$ = $ConstrT(H^{-1} \circ H \circ g(t'_2))$ = $ConstrT(h(t'_2))$ = $ConstrT(h'(t'_2))$.

It follows that $H'^{-1}((t''_1 = t''_2))$ = $h'((t'_1 = t'_2))$, therefore $H^{-1}((t''_1 = t''_2))$ is in B', thus concluding our proof that H' is an isomorphism.

To further show that H' is a $col\_isomorphism$ if H is a $col\_isomorphism$ we use an identical argument as in the proof of Lemma 1.3.48. □

Based on the above result, we can then show that terminating cs_chase sequences with $sk\_unit\_constraints$ lead to $col\_isomorphic$ results, as follows:

Theorem 1.3.62. *Let B be an sk_body and $\mathcal{C}$ a set of $sk\_unit\_constraints$. Let B_1 and B_2 be the results of two terminating cs_chase sequences with $\mathcal{C}$ over B.*

Then $\overline{B_1}$ and $\overline{B_2}$ are $col\_isomorphic$.

Proof. As we did when proving homomorphic equivalence for the results of two terminating cs_chase sequences, for every intermediate result S_t of the cs_chase sequence leading to B_1 we show inductively, the existence of an isomorphism between $\overline{S_t}$ and a part P of $\overline{B_2}$, based on Lemma 1.3.61. It follows that there exists an isomorphism h_1 between $\overline{B_1}$ and a part of $\overline{B_2}$.

Reversely, we show the existence of an isomorphism from $\overline{B_2}$ to a part of $\overline{B_1}$.

But *since isomorphisms are injective* it follows that B_1 and B_2 must have *the same number of tuple variables* (and relational atoms), therefore h_1 is an isomorphism between $\overline{B_1}$ and $\overline{B_2}$, which concludes our proof. □

1.3.4 The Provenance-Aware Chase

We will describe in this subsection the Provenance-Aware Chase, further denoted pa_chase. As is the case for the Standard Chase and the Conservative Chase, the pa_chase is an iterative procedure consisting in a sequence of steps. As announced in Section 1.2, the pa_chase essentially consists in *instrumenting the cs_chase with provenance*. pa_chase steps will thus take as input a *provenance-adorned sk_body* and an $sk\_unit\_constraint$[6] and yield as output a *provenance-adorned sk_body*.

1.3.4.1 Provenance formulae and provenance-adorned sk_bodies

To describe *provenance-adorned sk_bodies*, we will first introduce the concept of provenance formulae and their associated operations.

Definition 1.3.63 (Provenance formula)**.** *Given a finite set of symbols $\mathcal{P}$, called a* provenance vocabulary*, a provenance formula over $\mathcal{P}$ is either*

- True*, or*
- False*, or*
- *a boolean formula in DNF over the provenance symbols, using the *(AND) and +(OR) operators: $F = C_1 + ... + C_n$, where $C_i = S_i^1 * ... * S_i^{n_i}, S_i^j \in \mathcal{P}$ and C_i is called a* provenance conjunct*. We will further call the symbols in $\mathcal{P}$* provenance terms*.*

We denote by $ProvForms(\mathcal{P})$ the set of all provenance formulae over $\mathcal{P}$.

[6] According to the previous section, the results below can also be generalized to $sk\_constraints$ in general.

Note that we can view provenance conjuncts as subsets of P and provenance formulae as subsets of the power set of P, $\mathfrak{P}(\mathcal{P})$, such that $\mathrm{False} = \emptyset$ and $\mathrm{True} = \mathfrak{P}(\mathcal{P})$. We will hereafter use the standard set operations symbols with straightforward semantics for provenance formulae and provenance conjuncts.

We also define the *subsumption* of provenance formulae, as the reverse of logical implication:

Definition 1.3.64 (Provenance subsumption)**.** *A provenance formula F_1 over $\mathcal{P}$, $F_1 = C_1^1 + \cdots + C_n^1$ subsumes a provenance formula F_2 over $\mathcal{P}$, $F_2 = C_1^2 + \cdots + C_m^2$, denoted $F_1 \prec F_2$, iff $F_2 \longrightarrow F_1$, that is, iff $\forall i \in \{1, .., m\}\ \exists j \in \{1, .., n\}$ s.t. $C_j^1 \subseteq C_i^2$.*

It is easy to show that by definition of the subsumption, the following hold:

Lemma 1.3.65. *Let P_1 and P_2 be provenance formulae and P be a provenance conjunct. Then the following hold:*

1. *if $P_1 \prec P_2$ then $P_1 * P_2 = P_2$.*
2. *if $(P_1 * P_2) \prec P$, then $P_1 \prec P$ and $P_2 \prec P$.*
3. *if $(P_1 + P_2) \prec P$, then at least one of P_1 or $P_2 \prec P$.*

Using subsumption, we define the *reduced form* of a provenance formula:

Definition 1.3.66 (Reduced form of a provenance formula)**.** *Let F be a provenance formula over a vocabulary $\mathcal{P}$. We define the reduced form of F, $rf(F)$, as the formula $F' = C_1 + ... + C_m$ such that:*

1. $F' \subseteq F$
2. $F' \prec F$
3. $\forall i \neq j,\ C_i \not\prec C_j$ *and* $C_j \not\prec C_i$

The following lemma shows that the reduced form of a provenance formula is well defined and further provides an operational procedure for its computation

Lemma 1.3.67. *Let $F = C_1 + ... + C_m$ be a provenance formula over a vocabulary $\mathcal{P}$. Then $rf(F)$ is unique and computable by removing from F all conjuncts subsumed by other conjuncts.*

Proof. Suppose that there are two formulae $F_1 = C_1^1 + ... + C_n^1$ and $F_2 = C_1^2 + ... + C_p^2$ respecting the properties (1)-(3) that define the reduced form of F. We will show that $F_2 = F_1$.

Indeed, we will show that $F_2 \subseteq F_1$. Let C_i^2 be a conjunct in F_2. Since $F_2 \subseteq F$, it follows that $C_i^2 \in F$. But since $F_1 \prec F$, there exists C_j^1 s.t. $C_j^1 \subseteq C_i^2$. Furthermore, because $F_1 \subseteq F$, it follows that $C_j^1 \in F$. Then there exists $C_k^2 \in F_2$ s.t. $C_k^2 \subseteq C_j^1 \subseteq C_i^2$. But since according to the definition F_2 does not contain pairwise subsumed conjuncts, $i = k$ and $C_k^2 = C_j^1 = C_i^2$. It follows that $C_i^2 \in F_1$.

In a similar manner we show that $F_1 \subseteq F_2$. To conclude the proof of the lemma, it is straightforward to show that removing subsumed conjuncts from F leads to a provenance formula respecting the definition of the reduced form. □

We will use provenance formulae as *adornments* on sk_bodies as follows:

Definition 1.3.68 (Provenance adornment and *provenance-adorned sk_bodies*)**.** *Let $\mathcal{P}$ be a provenance vocabulary and B an sk_body. Let $Prov$ be a function defined on all atoms of B, with values in $ProvForms(\mathcal{P})$. Then $Prov$ is called a* provenance adornment *of B and the couple $(B, Prov)$ is called a* provenance-adorned sk_body.

We will hereafter refer to the values of $Prov$ on the atoms of B as the provenance *of the atoms of B.*

Provenance adornment of terms. The provenance adornment $Prov$ of an sk_body B induces a function $Prov_{terms}$ from $T(B)$ to $ProvForms(P)$, called the provenance adornment of terms, as follows:

1. $Prov_{terms}(r.A) = Prov(r \in R)$, for a projection term.
2. $Prov_{terms}(K) = \mathrm{True}$, for a constant
3. $Prov_{terms}(f(a_0, \ldots, a_n)) = \prod Prov_{terms}(a_i)$ for a Skolem term (and $Prov_{terms}(f()) = \mathrm{True}$, for a Skolem term with no argument)

Since there is no ambiguity, and in order to avoid clutter, we will use in the following the notation $Prov$ to also denote $Prov_{terms}$.

Provenance adornment of the closed version. Based on the provenance adornment $Prov$ of an sk_body B and the induced provenance adornment of terms, we can further define a provenance adornment $\overline{Prov}$ on $\overline{B}$, as follows:

1. for a relational atom $(r \in R) \in [\overline{B}]_{rel}$, $\overline{Prov}(r \in R) = Prov(r \in R)$
2. for a constructive equality atom $(t_1 = t_2) \in [\overline{B}]_{constr\_eq}$, $\overline{Prov}(t_1 = t_2)$ =$Prov(t_1 = t_2)$
3. for an equality atom $(t_1 = t_2) \in [\overline{B}]_{eq}$:
   (a) if $t_1 = t_2$, then $\overline{Prov}(t_1 = t_2)$ = True
   (b) else, we define a *simple path* sp between t_1 and t_2 as follows: let $s_0 = t_1, s_1, \ldots, s_n = t_2$ be an ordered subset of $T(B)$, such that the equality $(s_i = s_{i+1})$ or its symmetrical is in $[B]_{eq}$ or in $[B]_{constr\_eq}$.
   Let $p_i = Prov((s_i = s_{i+1}))$ if $(s_i = s_{i+1})$ is in $[B]_{eq}$ or in $[B]_{constr\_eq}$, respectively $Prov((s_{i+1} = s_i))$ if $(s_{i+1} = s_i)$ is in $[B]_{eq}$ or in $[B]_{constr\_eq}$.We denote by $Prov_{path}(sp)$ the product $Prov(s_1) * \cdots * Prov(s_{n-1}) * p_i * \cdots * p_n$. Note that the above product includes all the adornments of equality atoms on the path as well as all the adornments on the terms on the path *except for its extremities*.
   We denote by $SP(t_1 = t_2)$ the set of all simple paths between t_1 and t_2. Then $\overline{Prov}(t_1 = t_2) = \sum_{sp \in SP(t_1 = t_2)} Prov_{path}(sp)$

Provenance of a set of atoms. Based on the provenance of the atoms in an $sk\_body\ B$, we further define the provenance of any subset of atoms B' of B as the product of the provenance of all the atoms in the set (since there is no ambiguity, we will use the same notation $Prov$).

$$Prov(B') = \prod_{a \in B'} Prov(a)$$

Full provenance of an atom. Starting from a provenance adornment of an $sk\_body\ B$, we further define a function $Prov_{full}$ on all atoms of B, as follows:

1. $Prov_{full}(r \in R) = Prov(r \in R)$, for $r \in R$ a relational atom in $[B]_{rel}$
2. $Prov_{full}(t_1 = t_2)$ = $Prov(t_1) * Prov(t_2) * Prov(t_1 = t_2)$, for $(t_1 = t_2)$ an equality in $[B]_{eq}$ or in $[B]_{constr\_eq}$.

We will call the values of $Prov_{full}$ on an atom the *full provenance* of the atom. Note that the full provenance of an equality atom *is in general different from its adornment*. Note also that for equality atoms this is the notion of provenance we implicitly refer to in Section 1.2, when stating invariant ($\diamond$) and goal ($\dagger$).

We extend in a straightforward manner the notion of full provenance to a set of atoms: for B' a set of atoms,

$$Prov_{full}(B') = \prod_{a \in B'} Prov_{full}(a)$$

1.3.4.2 Provenance-Aware Chase steps and sequences

We are now ready to formally define the pa_chase steps, by first listing their *conditions of application* and then by specifying their application, i.e. how they produce an output *provenance-adorned sk_body* from an input *provenance-adorned sk_body*.

Definition 1.3.69 (pa_chase step conditions of application)**.** *A pa_chase step with $sk\_unit\_constraint\ C$ on a* provenance-adorned sk_body $(B, Prov)$ *applies iff:*

1. *There exists a homomorphism h from C_{prem} to $\overline{B}$*
2. *Either:*
   (a) *there exists no homomorphism h' compatible with h from C_{concl} to $\overline{B}$, or*
   (b) *for any such h', $\overline{Prov}(h'(C_{concl})) \not< \overline{Prov}(h(C_{prem}))$*

Definition 1.3.70 (pa_chase step application)**.** *Applying a pa_chase step with $sk\_unit\_constraint$ C on a* provenance-adorned sk_body $(B, Prov)$*, given homomorphism h from C_{prem} to B, results in a new* provenance-adorned sk_body $(B', Prov') = Pa\_Chase\_Step\_Res((B, prov), C, h)$*, such that:*

The $sk\_body\ B' \supseteq B$ is obtained as follows:

1. *if no homomorphism compatible with h exists, let $B' = CS\_Chase\_Step\_Res(B, C, h)$ and let h' be the corresponding cs_chase step compatible homomorphism.*

2. *else, let h' a homomorphism compatible with h and let $B' = B \cup h'(C_{concl})$.*

The provenance adornment of B', $Prov$', is obtained as follows. Let $P_{prem} = \overline{Prov}(h(C_{prem}))$. Then:

1. *for constructive equalities in $B' - B$, $Prov$' =* True

2. *for relational atoms and non-constructive equalities in $B' - B$, $Prov$' = P_{prem}*

3. *for relational atoms and non-constructive equalities in $B - h'(C_{concl} - [C_{prem}]_{rel})$, $Prov$' = $Prov$*

4. *for relational atoms and non-constructive equalities in $B \cap h'(C_{concl} - [C_{prem}]_{rel})$, $Prov$' = $Prov$+P_{prem}.*

Provenance enriching and atom creation steps. Let us take a closer look at the definition of a pa_chase step.

1. If no $homomorphism$ from C_{concl} to $\overline{B}$ exists, then the provenance adornment stays the same on B and the pa_chase step will introduce all the atoms in h'(C_{concl} - $[C_{prem}]_{rel}$) with fresh adornments, True in the case of constructive equalities and P_{prem} otherwise. We call such step an atom creation step.

2. else

   (a) if the pa_chase step is with an sk_TGD, then $B' = B$ (according to the closure definition) and for the unique relational atom in h'($[C_{concl}]_{rel}$ - $[C_{prem}]_{rel}$), its new adornment $Prov$' will be the sum of the old adornment $Prov$ and P_{prem}. We will call such step a provenance enriching step.

   (b) else, for an sk_EGD, according to whether h'($[C_{concl}]_{eq}$) is or not in B, B' can be equal to B (and the adornment of the equality in B enriched with P_{prem} as above) or contain an additional equality atom, with fresh adornment P_{prem}. Although technically speaking the latter case involves an atom addition to B, the atom is already *in the closed version of B*. We will thus also call this type of pa_chase step a *provenance enriching step*, and keep in mind that a provenance-enriching step with an equality addition may be possible only in the case of an sk_EGD.

As was the case for the standard chase and the cs_chase, it is easy to show that the function h' constructed in the pa_chase step application on a *provenance-adorned sk_body* $(B, Prov)$ is a $homomorphism$ compatible with h, from C_{concl} to B'. Similar to the case of standard chase and cs_chase steps, we will hereafter call h' the pa_chase step compatible $homomorphism$.

Provenance-Aware Chase sequences. Given an *provenance-adorned sk_body* $(B, Prov)$ and a set of $sk\_unit\_constraints$ $\mathcal{C}$, a pa_chase sequence consists in producing the *provenance-adorned sk_bodies* $(B_0, Prov_0)$, $(B_1, Prov_1)$, ..., such that:

1. $(B_0, Prov_0) = (B, Prov)$

2. $(B_i, Prov_i)$ is obtained from $(B_{i-1}, Prov_{i-1})$ by the following operations:

   (a) pick $C \in \mathcal{C}$ s.t. a pa_chase step with C applies on $(B_{i-1}, Prov_{i-1})$, with a $homomorphism\ h$ from C_{prem} to $\overline{B_{i-1}}$;

   (b) let $(B_i, Prov_i)$:=$Pa\_Chase\_Step\_Res((B_{i-1}, Prov_{i-1}), C, h)$;

For a finite pa_chase sequence with a number of steps k, we denote by the *result* of the sequence the *provenance-adorned sk_body* $(B_k, Prov_k)$ produced by the last step.

A *full* pa_chase sequence consists in applying pa_chase steps as long as there exists at least an $sk\_unit\_constraint\ C \in \mathcal{C}$ such that a pa_chase with C applies. A *terminating* pa_chase sequence is a full pa_chase sequence that terminates after a finite number of steps n – that is, $(B_n, Prov_n)$ is such that for any $sk\_unit\_constraint\ C$ in $\mathcal{C}$, and any possible $homomorphism$ h from C_{prem} to $\overline{B_n}$, there exists a compatible $homomorphism\ h'$ from C_{concl} to $\overline{B_n}$ such that furthermore $\overline{Prov}_n(h'(C_{concl})) \prec \overline{Prov}_n(h(C_{prem}))$

1.3.4.3 The Provenance Pick, the Provenance-Aware Chase and the Conservative Chase

We dedicate this subsection to showing the essential link between the Conservative Chase and the Provenance-Aware Chase, via the *Provenance Pick* operation.

The Provenance Pick allows retrieving sk_bodies from *provenance-adorned sk_bodies*, by selecting all the atoms whose full provenance is implied by a conjunct, as follows:

Definition 1.3.71 (Provenance Pick)**.** *Let $(B, Prov)$ be a provenance-adorned sk_body over a provenance vocabulary $\mathcal{P}$ and $P \subseteq \mathcal{P}$ a provenance conjunct. We define $Pick(P, (B, Prov))$ as the sk_body $B' \subseteq B$, obtained as follows:*

1. $[B']_{rel}$ = *{$r \in R$}, such that $(r \in R) \in [B]_{rel}$ and $Prov_{full}(r \in R) \prec P$.*

2. $[B']_{constr\_eq}$ = *{$t_1 = t_2$}, such that $(t_1 = t_2) \in [B]_{constr\_eq}$ and $Prov_{full}(t_1 = t_2) \prec P$.*

3. $[B']_{eq}$ = *{$t_1 = t_2$}, such that $(t_1 = t_2) \in [B]_{eq}$ and $Prov_{full}(t_1 = t_2) \prec P$.*

One can easily show, given the definition of the full provenance of an atom, that applying the Pick operation on a *provenance-adorned sk_body* results indeed in an sk_body.

Remember that we have stated that the Provenance-Aware Chase is essentially the Conservative Chase with provenance annotations. In the following we will show that over provenance-adorned $bodies$, the two procedures *commute* via the Pick operation, as follows:

Theorem 1.3.72. *Let B be a body and $Prov$ a provenance adornment of B over a provenance vocabulary $\mathcal{P}$. Let $\mathcal{C}$ be a set of $sk\_unit\_constraints$. Let $(B', Prov')$ be the result of a terminating pa_chase sequence with $\mathcal{C}$ over $(B, Prov)$.*

Let $P \subseteq \mathcal{P}$ be a provenance conjunct. Let $B_p = Pick(P, (B, Prov))$ and B'_p be the result of a terminating cs_chase sequence with $\mathcal{C}$ over B_p.

Then $\overline{B'_p}$ and $\overline{Pick(P, (B', Prov'))}$ are $col\_isomorphic$.

To prove the above, we will show several important results regarding the pa_chase and the Provenance Pick. We start by noting that the following holds, as a direct consequence of the definition of provenance of the closure and full provenance:

Proposition 1.3.73. *Let $(B, Prov)$ be a* provenance-adorned sk_body *and $(t_1 = t_2)$ an equality in $[\overline{B}]_{eq}$, such that $t_1 \neq t_2$.*

Then $\overline{Prov}_{full}((t_1 = t_2))$ = $\sum_{sp \in SP(t_1 = t_2)} \prod Prov_{full}(eq_i),\ where\ eq_i \in sp$

Using Proposition 1.3.73, we show that the Provenance Pick *commutes* with the closure, as follows:

Proposition 1.3.74. *Let $(B, Prov)$ be a* provenance-adorned sk_body *over a provenance vocabulary $\mathcal{P}$.*

Let $P \subseteq \mathcal{P}$ be a provenance conjunct.

Then $Pick(P, (\overline{B}, \overline{Prov}))$ = $\overline{Pick(P, (B, Prov))}$

Proof. It is straightforward to show that $[Pick(P, (\overline{B}, \overline{Prov}))]_{rel}$ = $[\overline{Pick(P, (B, Prov))}]_{rel}$. Indeed, $\overline{Prov}$(a) = $Prov$(a) for every relational atom in $[B]_{rel}$ = $[\overline{B}]_{rel}$. We apply the same reasoning for constructive equalities.

We further show that $[Pick(P, (\overline{B}, \overline{Prov}))]_{eq}$ = $[\overline{Pick(P, (B, Prov))}]_{eq}$.

Indeed, let $(t_1 = t_2)$ be an equality in $[Pick(P, (\overline{B}, \overline{Prov}))]_{eq}$. Then $\overline{Prov}_{full}((t_1 = t_2)) \prec P$. But according to the properties of subsumption (Lemma 1.3.65) and Proposition 1.3.73 it follows that there exists a simple path sp from t_1 to t_2 such that for every equality $eq_i \in sp$, $Prov_{full}(eq_i) \prec P$. Then sp is a simple path in $\overline{Pick(P, (B, Prov))}$ and $(t_1 = t_2)$ is in $\overline{Pick(P, (B, Prov))}$.

Reversely, for an equality $(t_1 = t_2)$ in $[\overline{Pick(P, (B, Prov)}]_{eq}$, *every simple path* will consist of equalities eq_i such that $Prov_{full}(eq_i) \prec P$. Then $\overline{Prov}_{full}((t_1 = t_2)) \prec P$, therefore $(t_1 = t_2)$ is in $Pick(P, (\overline{B}, \overline{Prov}))$, which concludes our proof. □

We continue by showing that in a pa_chase sequence starting from a $body$, the provenance of a term is *always subsumed* by the provenance of its constructive term:

Proposition 1.3.75. *Let B_0 be a $body$ and $Prov_0$ an adornment of B_0 over a provenance vocabulary $\mathcal{P}$. Let $\mathcal{C}$ be a set of $sk\_unit\_constraints$ and $(B', Prov')$ be a* provenance-adorned sk_body *resulting from a pa_chase sequence over $(B_0, Prov_0)$ with $\mathcal{C}$.*

Then for every term t in $T(B')$, $Prov'(ConstrT(t)) \prec Prov'(t)$

Proof. We will show by induction on the pa_chase steps that the above holds. It is clearly the case for the initial $(B_0, Prov_0)$, since B_0 is a $body$ and thus $ConstrT(t) = t$ for all terms.

Let $(B, Prov)$ be the result preceding $(B', Prov')$ in the pa_chase sequence. Assuming the result holds for $(B, Prov)$ we will show that the result holds for $(B', Prov')$. Indeed, let C be the $sk\_unit\_constraint$ corresponding to the pa_chase step leading from $(B, Prov)$ to $(B', Prov')$.

If C is an sk_EGD then the above result is straightforward. Indeed, by definition, pa_chase steps with sk_EGDs do not change the provenance of terms.

If C is an sk_TGD, then let h be the $homomorphism$ from C_{prem} to $\overline{B}$ corresponding to the pa_chase step and h' be the pa_chase step compatible $homomorphism$. We will show that the above property is verified for projection terms in B'. By definition of the constructive terms and provenance of Skolem terms, it then extends directly to all terms of B'.

If the pa_chase step is an atom creation step, then for every projection term t_1 in $T(B') - T(B)$, $ConstrT(t_1) = ConstrT(h(t'_2))$ where $t'_2 \in T(C_{prem})$ and $h(t'_2) \in B$. According to our induction hypothesis, $Prov(ConstrT(h(t'_2)))$ then subsumes $Prov(h(t'_2))$. On the other hand, $h(t'_2) \in T(h(C_{prem}))$, therefore $Prov(h(t'_2)) \prec \overline{Prov}(h(C_{prem}))$. But since the pa_chase step is an atom creation step $Prov'(t_1) = Prov(h(C_{prem}))$. It then follows that $Prov(ConstrT(h(t'_2))) \prec Prov'(t_1)$, so $Prov'(ConstrT(t_1)) \prec Prov'(t_1)$

For a provenance enriching step, the constructive equalities in $h'(C_{concl})$ already exist in B. Furthermore, by definition of the pa_chase step, for each projection term t_1 defined by such constructive equality, $Prov'(t_1) = Prov(t_1) + \overline{Prov}(h(C_{prem}))$. On the other hand, since these equalities exist in B, by induction hypothesis $Prov(ConstrT(t_1))) \prec Prov(t_1)$.

Further, as in the case above, we can show that $Prov(ConstrT(t_1)) \prec \overline{Prov}(h(C_{prem}))$. Since $Prov'(ConstrT(t_1)) = Prov(ConstrT(t_1))$, accordingly, $Prov'(ConstrT(t_1)) \prec \overline{Prov}(h(C_{prem}))+Prov(t_1)$, therefore $Prov'(ConstrT(t_1)) \prec Prov'(t_1)$, which concludes our proof. □

A direct consequence of the above proposition is that in a pa_chase sequence starting from a provenance adorned $body$, *the full provenance of a constructive equality is always equal to the provenance of the term it defines*:

Proposition 1.3.76. *Let* $(B, Prov)$ *be a* provenance-adorned sk_body *resulting from a* pa_chase *sequence over a provenance-adorned* $body$.

Then for every constructive equality $(t = t')$ *in* $(B, Prov)$, $Prov_{full}(t = t') = Prov(t)$

Indeed, it is enough to notice that all introduced constructive equalities are annotated with True, by definition of a pa_chase step. The result above then follows from Proposition 1.3.75 and Lemma 1.3.65. This result further allows us to state the following:

Lemma 1.3.77. *Let* $(B, Prov)$ *be a* provenance-adorned sk_body *resulting from a* pa_chase *sequence over a provenance-adorned* $body$.

Let h *be a* $homomorphism$ *from an* sk_body D *to* $(B, Prov)$.

Then $Prov(h(D)) = Prov_{full}(h(D))$.

Proof. For regular equalities notice that the corresponding projection terms must be in relational atoms in $h(D)$, therefore the provenance of their end points is already included in the provenance of those atoms. For constructive equalities in $h(D)$ we rely on Proposition 1.3.76 and we further note as in the previous case that the provenance of the terms those constructive equalities define is already included in the provenance of the respective relational atoms. □

By mixing the above result with Proposition 1.3.74, we can infer the following:

Lemma 1.3.78. *Let* $(B, Prov)$ *be a* provenance-adorned sk_body *resulting from a* pa_chase *sequence over a provenance-adorned body.*

Let P *be a provenance conjunct.*

Let h *be a* $homomorphism$ *from an* sk_body D *to* $(\overline{B}, \overline{Prov})$ *such that* $\overline{Prov}(h(D)) \prec P$. *Then* h *is a* $homomorphism$ *from* D *to* $\overline{Pick(P, (B, Prov))}$.

Reversely, let h *be a* $homomorphism$ *from an* sk_body D *to* $\overline{Pick(P, (B, Prov))}$. *Then* h *is a* $homomorphism$ *from* D *to* $(\overline{B}, \overline{Prov})$ *such that* $\overline{Prov}(h(D)) \prec P$.

Equally important, based on Proposition 1.3.76, we can note that *if a relational atom is picked according to a conjunct, then all its constructive equalities are picked*. Putting together all the above results we can then provide the following *essential characterization of a* pa_chase *step and the Provenance Pick*:

Lemma 1.3.79. *Let* $(B, Prov)$ *be a* provenance-adorned sk_body *resulting from a* pa_chase *sequence over a provenance-adorned body with a set of* $sk\_unit\_constraints$ $\mathcal{C}$.

Let C *be an* $sk\_unit\_constraint$ *such that a* pa_chase *step with* C *applies on* $(B, Prov)$ *with* $homomorphism$ h *from* C_{prem} *to* $\overline{B}$, *yielding* $(B', Prov')$
= $Pa\_Chase\_Step\_Res((B, Prov), C, h)$. *Let* $P_{prem} = \overline{Prov}(h(C_{prem}))$.

Let $P \subseteq \mathcal{P}$ *be a provenance conjunct.*

Let $B_p = Pick(P, (B, Prov))$ *and* $B'_p = Pick(P, (B', Prov'))$.

Then:

1. *if* $P_{prem} \not\prec P$ *then* $B'_p = B_p$.
2. *else,* h *is a* $homomorphism$ *from* C_{prem} *to* $\overline{B_p}$, h' *is a* $homomorphism$ *compatible with* h *from* C_{concl} *to* B'_p *and furthermore:*
   (a) *either* $B'_p = B_p$, *or*
   (b) $B'_p = B_p \cup h'(C_{concl})$ *and* $B_p \cap h'(C_{concl}) = h([C_{prem}]_{rel})$. *Furthermore, let* H *be an isomorphism from* $\overline{B_p}$ *to a part* J *of an* sk_body D. *Let* $g = H \circ h$ *be the corresponding* $homomorphism$ *from* C_{prem} *to* J.
   If there exists a $homomorphism$ g' *compatible with* g *from* C_{concl} *to* D, *then there exists an isomorphism* H'' *from* B'_p *to* $J \cup g'(C_{concl})$ *such that, moreover, if* H *is a* $col\_isomorphism$ *then* H'' *is a* $col\_isomorphism$.

Proof. Let $P_{prem} = \overline{Prov}(h(C_{prem}))$.

We start by noting that for any atom a in B', $Prov_{full}{}'(a)$ is either $Prov_{full}(a)$ or $Prov_{full}(a)$+P_{prem}. Indeed, while for an sk_TGD it is a direct consequence of the pa_chase step and Proposition 1.3.76, for an sk_EGD it follows directly by the fact that for the unique equality $(t_1 = t_2)$ in $h'(C_{concl})$, $Prov(t_1) \prec P_{prem}$ and $Prov(t_2) \prec P_{prem}$.

We can then conclude according to Lemma 1.3.65 that if an atom a is in $Pick(P, (B', Prov'))$ then at least one of $Prov_{full}(a)$ or P_{prem} subsumes P. If $P_{prem} \not\prec P$, then $Prov_{full}(a)$ must subsume P, and so a must be in $Pick(P, (B, Prov))$, thus proving the first point above.

Continuing, if $P_{prem} \prec P$, then according to Lemma 1.3.78 it follows directly that h is a $homomorphism$ from C_{prem} to $\overline{B_p}$. Furthermore, relying on the above and the definition of a

pa_chase step, it is easy to show that for every atom a in $h'(C_{concl})$, ${Prov_{full}}'(a) \prec P_{prem}$. It then follows that h' is a $homomorphism$ from C_{concl} to B'_p.

Furthermore, if $B'_p \neq B_p$, it is straightforward to show that the unique equality (in the case of an sk_EGD) or the unique relational atom and its constructive equalities (in the case of an sk_TGD) in $h'(C_{concl} - [C_{prem}]_{rel})$ are respectively disjoint from B_p. The rest of the proof of point $2.b$ is then identical to the proof of Lemma 1.3.61. □

Note the extreme similarity of the last point of the lemma above with its analogous statements for cs_chase steps. We basically show that on the picked versions of the *provenance-adorned sk_bodies* the pa_chase behaves exactly like the cs_chase. Accordingly, we will prove Theorem 1.3.72 in the same manner than Theorem 1.3.62. Indeed, we will first show that the following holds:

Lemma 1.3.80. *Let B be a body and $Prov$ an adornment of B over a provenance vocabulary $\mathcal{P}$. Let $\mathcal{C}$ be a set of sk_unit_constraints. Let $P \subseteq \mathcal{P}$ be a provenance conjunct.*

Let $(B', Prov')$ be the result of a terminating pa_chase sequence with $\mathcal{C}$ over $(B, Prov)$.

Let $B_0 = Pick(P, (B, Prov)), B_1, \ldots$ be a cs_chase sequence with $\mathcal{C}$ over $Pick(P, (B, Prov))$.

Then there exists a $col\_isomorphism$ from $\overline{B_i}$ to a part of $\overline{Pick(P, (B', Prov'))}$.

Proof. The proof of the above is very similar to the proof of existence of a $col\_isomorphism$ from the intermediate results of a cs_chase sequence to a part of the result of a terminating cs_chase sequence. Indeed, since $B_0 = Pick(P, (B, Prov)) \subseteq Pick(P, (B', Prov'))$ we can exhibit the $col\_isomorphism$ $h_0 = Id$.

Assuming that a $col\_isomorphism$ h_t exists from $\overline{B_t}$ to a part of $\overline{Pick(P, (B', Prov'))}$, we will show the existence of a $col\_isomorphism$ h_{t+1} from $\overline{B_{t+1}}$ to a part of $\overline{Pick(P, (B', Prov'))}$ based on Lemma 1.3.61.

It is enough to notice that for h a $homomorphism$ from C_{prem} to $\overline{B_t}$, $g = h_t \circ h$ is a $homomorphism$ from C_{prem} to $\overline{Pick(P, (B', Prov'))}$. But according to Lemma 1.3.78, g is then a $homomorphism$ from C_{prem} to $(\overline{B'}, \overline{Prov}')$ such that $\overline{Prov}(h(C_{prem})) \prec P$. Then since $(B', Prov')$ is the result of a terminating pa_chase sequence it follows that there must exist at least one $homomorphism$ g' compatible with g, from C_{concl} to $(\overline{B'}, \overline{Prov}')$, such that $\overline{Prov}(g'(C_{concl})) \prec \overline{Prov}(g(C_{prem}))$.

Then $\overline{Prov}(g'(C_{concl})) \prec P$ and according to Lemma 1.3.78 g' is a $homomorphism$ compatible with g from C_{concl} to $\overline{Pick(P, (B', Prov'))}$. We are then in the conditions of Lemma 1.3.61 and it follows that there exists a $col\_isomorphism$ from B_{t+1} to a part of $Pick(P, (B', Prov'))$, and accordingly from $\overline{B_{t+1}}$ to a part of $\overline{Pick(P, (B', Prov'))}$, thus concluding our proof. □

Symetrically, we will show that the following holds:

Lemma 1.3.81. *Let B be a body and $Prov$ an adornment of B over a provenance vocabulary $\mathcal{P}$. Let $\mathcal{C}$ be a set of sk_unit_constraints. Let $P \subseteq \mathcal{P}$ be a provenance conjunct.*

Let B' be the result of a terminating cs_chase sequence with $\mathcal{P}$ over $Pick(P, (B, Prov))$.

Let $(B_0 = B, Prov_0 = Prov), (B_1, Prov_1), \ldots$ be a pa_chase sequence with $\mathcal{C}$ over $(B, Prov)$.

Then there exists a $col\_isomorphism$ *from* $Pick(P, (B_i, Prov_i))$ *to a part of* B'.

The proof of the above results is quasi-identical to the proof of Lemma 1.3.61, further relying on Lemma 1.3.79.

Relying on the two above results, we can then prove Theorem 1.3.72 in an identical fashion to Theorem 1.3.62 (stating isomorphism between results of cs_chase sequences with $sk\_unit\_constraints$), by relying on the injective nature of the exhibited isomorphisms in Lemmas 1.3.80 and 1.3.81.

1.3.4.4 Termination of the Provenance-Aware Chase

Based on Lemmas 1.3.80 and 1.3.81 we can further tightly link the termination behaviour of the pa_chase and the cs_chase as follows:

Theorem 1.3.82. *Let* B *be a body and* $Prov$ *an adornment of* B *over a provenance vocabulary* $\mathcal{P}$. *Let* C *be a set of* $sk\_unit\_constraints$. *Then:*

1. *if there exists a terminating* cs_chase *sequence of* B *with* $\mathcal{C}$, *then all* pa_chase *sequences of* $(B, Prov)$ *with* $\mathcal{C}$ *terminate.*
2. *reversely, if there exists a terminating* pa_chase *sequence of* $(B, Prov)$ *with* $\mathcal{C}$ *then all full* cs_chase *sequences with* $\mathcal{C}$ *over* B *terminate.*

Proof. Indeed, it is enough to consider the provenance conjunct $P = \mathcal{P}$.

We will start by proving point (2) above. Let $(B_n, Prov_n)$ be the result of a terminating pa_chase sequence with $\mathcal{C}$ over $(B, Prov)$. Since $P = \mathcal{P}$ it follows that $B = Pick(P, (B, Prov))$ and $B_n = Pick(P, (B_n, Prov_n))$.

Let $B_0 = B, B_1, \ldots$ be a cs_chase sequence with $\mathcal{C}$ over B. Then it is also a cs_chase sequence with $\mathcal{C}$ over $Pick(P, (B, Prov))$. Accordingly, by Lemma 1.3.80, there exists a $col\_isomorphism$ between $\overline{B_i}$ and a part of $\overline{B_n}$. Since isomorphisms are injective, it follows that there exists k such that no more relational atom creation occurs in the cs_chase sequence starting from B_k. Then it follows that there may be only a limited number of equality atom creation steps (similar to the proof of Theorem 1.3.49) and the cs_chase sequence terminates.

To further prove point (1), let B_m be the result of a terminating cs_chase sequence over B. Then B_m is also the result of a terminating cs_chase sequence over $Pick(P, (B, Prov))$.

On the other hand, let $(B_0 = B, Prov_0 = Prov), (B_1, Prov_1), \ldots$ be a pa_chase sequence with $\mathcal{C}$ over $(B, Prov)$. Then according to Lemma 1.3.81, there exists a $col\_isomorphism$ from $B_i = Pick(P, (B_i, Prov_i))$ to a part of B_m. But then since isomorphisms are injective it follows that there exists m_1 such that all pa_chase steps after m_1 cannot be atom creation steps. As a consequence, starting from $(B_{m_1}, Prov_{m_1})$, all pa_chase steps are provenance enriching steps. On the other hand, recall that the provenance enriching steps can only add some provenance on atoms of $\overline{B_{m_1}}$. The number of such atoms is finite. Furthermore, since a provenance enriching step adds at least one provenance conjunct and the number of provenance conjuncts in $\mathcal{P}$ is finite, it follows that there can only be a finite number of provenance enriching steps, therefore the pa_chase sequence terminates. □

Remark. While the fact of exhibiting $col\_homomorphisms$ in the case of the cs_chase with $sk\_constraints$ that are not necessarily $sk\_unit\_constraints$ allowed us to derive strong termination equivalence, the isomorphisms exhibited above, whether among cs_chase sequences with $sk\_unit\_constraints$ or cs_chase sequences and pa_chase sequences, are enough to state the strong termination links existing among such sequences. The fact that the isomorphisms in question map in fact collapsible atoms can be seen as merely a "bonus" or alternative criteria for termination.

Weakly acyclic constraints. Theorem 1.3.82, together with Theorem 1.3.56 (stating the termination of the cs_chase for weakly acyclic constraints) allow us to further derive termination of pa_chase sequences in the case of weakly acyclic constraints, as follows:

Theorem 1.3.83. *Let B be a $body$ and $Prov$ a provenance adornment on B over a provenance vocabulary $\mathcal{P}$. Let $\mathcal{C}$ be a set of weakly acyclic constraints.*

Then all full pa_chase sequences on $(B, Prov)$ with $skunit(\mathcal{C})$ terminate.

1.3.4.5 The Provenance-Aware Chase and the Standard Chase

Based on the results linking the the Provenance-Aware Chase and the Conservative Chase, and the equivalence between the cs_chase and the Standard Chase on $bodies$, we can further infer that the Standard Chase and the pa_chase *commute via the Provenance Pick* as follows:

Theorem 1.3.84. *Let B be a $body$ and $Prov$ an adornment of B over a provenance vocabulary $\mathcal{P}$. Let $\mathcal{C}$ be a set of constraints.*

Let $P \subseteq \mathcal{P}$ be a provenance conjunct. Let B'_p be the result of a terminating standard chase sequence with $\mathcal{C}$ on $Pick(P, (B, Prov))$.

Let $(B', prov')$ be the result of a terminating pa_chase sequence with $skunit(\mathcal{C})$ on $(B, prov)$.

Then $\overline{B'_p}$ and $Body(\overline{Pick(P, (B', prov'))})$ are homomorphically equivalent.

The above result provides the *correctness basis* for our reformulation algorithm $Prov_{C\&B}$, as we will see in the following section.

1.3.5 The Provenance-Aware Chase & Backchase

We have presented in Section 1.2 an overall view of our reformulation algorithm, the Provenance-Aware Chase & Backchase ($Prov_{C\&B}$). Based on the concepts and statements of the previous subsections, we are now ready to detail this algorithm and prove its soundness and completeness. We start by detailing some terminology that allows us to fully explicitate $Prov_{C\&B}$.

Universal plan and universal body. Given a query Q formulated against a source schema S, a set of constraints $\mathcal{C}$ and a target schema T, we denote by $Q^{\mathcal{C}}$ the result of a terminating standard chase sequence over Q with $\mathcal{C}$.

We further denote by B_U and call the *universal body*, the restriction of $\overline{body(Q^{\mathcal{C}})}$ to the target schema T (recall that this restriction means that we consider the maximal sub-body of $\overline{body(Q^{\mathcal{C}})}$ induced by the relational atoms in T). Note that B_U is a *closed body*, that is, $\overline{B_U}$ =

B_U. We will further call the query $U = Query(Head(Q), B_U)$ the *universal plan*. For every subquery sq of U we denote by B_{sq} its corresponding $body$. Note that $B_{sq} \subseteq B_U$. We recall below the properties of the universal plan, restating them using $bodies$:

Proposition 1.3.85. *Let Q be a query formulated against a source schema S, T a target schema, $\mathcal{C}$ a set of weakly acyclic constraints (including the relationship between S and T) and U the corresponding universal plan. Then:*

1. *every reformulation of Q over T given $\mathcal{C}$ is (isomorphic to) a subquery sq of the universal plan*
2. *a subquery sq of the universal plan is a reformulation of Q iff there exists a $homomorphism$ from $body(Q)$ to $\overline{B^{\mathcal{C}}_{sq}}$, where $B^{\mathcal{C}}_{sq}$ is the result of a terminating Standard Chase sequence with $\mathcal{C}$ over B_{sq}.*

Canonical provenance vocabulary. Let B_U be a universal body. We denote by $\mathcal{P}_U$, and call the *canonical provenance vocabulary*, the provenance vocabulary containing as terms all the tuple variables of B_U (and therefore U): r_i is in $\mathcal{P}_U$ iff $r_i \in R_i$ is in B_U.

For every conjunct $C_i \subseteq \mathcal{P}_U$ we further denote by $sq(C_i)$ the corresponding subquery of U induced by C_i and by $B_{sq}(C_i)$ the corresponding sub-body of B_U. Reversely, we denote by $C(sq)$ the conjunct corresponding to a universal plan subquery. Note that all these are one-to-one correspondences between subqueries (and their corresponding $bodies$) and provenance conjuncts over $\mathcal{P}_U$.

Canonical adornment of the universal body. We denote by $Prov_U$ the provenance adornment of B_U over $\mathcal{P}_U$ obtained as follows:

1. $Prov_U((r_i \in R_i)) = r_i$, for a relational atom in $[B_U]_{rel}$
2. $Prov_U((t_1 = t_2))$ = True, for an equality atom in $[B_U]_{eq}$

We will call $Prov_U$ the *canonical adornment* of B_U. It is easy to show that the following holds:

Proposition 1.3.86. *Let $C_i \subseteq \mathcal{P}_U$ be a conjunct over the canonical provenance vocabulary. Then $B_{sq(C_i)}$ = $Pick(C_i, (B_U, Prov_U))$*

Using the notation above, we are now ready to present our reformulation algorithm $Prov_{C\&B}$:

$Prov_{C\&B}$ (Query Q, source schema S, target schema T, set of weakly acyclic constraints $\mathcal{C}$)

1 $B_U \leftarrow Universal\ Body(Q, \mathcal{C}, S, T)$
2 $Prov_U \leftarrow$ canonical adornment of B_U.
3 $(B', Prov') \leftarrow$ the result of a terminating pa_chase sequence over $(B_U, Prov_U)$ with $skunit(\mathcal{C})$.
4 $\mathcal{H} \leftarrow \{h,\ h\ is\ a\ homomorphism\ from\ body(Q)\ to\ (\overline{B'}, \overline{Prov'})\}$
5 $F \leftarrow \sum_{h \in \mathcal{H}} \overline{Prov'}(h(body(Q))$
6 **for** $C_i \in rf(F)$
7 **do**
8 **return** $sq(C_i)$

We further claim the soundness and completeness of $Prov_{C\&B}$, as follows:

Theorem 1.3.87. *Let Q be a SFW query formulated over a source schema S, T a target schema and $\mathcal{C}$ a set of weakly acyclic constraints over $S \cup T$*

Then the algorithm $Prov_{C\&B}$ is sound and complete, that is, it returns all and precisely *the minimal reformulations of Q against T given $\mathcal{C}$.*

Proof. The soundness and completeness of $Prov_{C\&B}$ can be restated as follows: a subquery sq of U is a minimal reformulation of Q under $\mathcal{C}$ iff $C(sq) \in rf(F)$. To prove this statement, we first show that the following hold:

1. Let C be a conjunct in F. Then $sq(C)$ is a reformulation of Q.

   Indeed, by definition of F, there exists h a $homomorphism$ from $body$(Q) to $(\overline{B'}, \overline{Prov}')$, such that $C \in \overline{Prov}'(h(body(Q)))$. It follows that $\overline{Prov}'(h(body(Q))) \prec C$. Then by Lemma 1.3.78, h is a $homomorphism$ from $body$(Q) to $\overline{Pick(C, (B', prov'))}$. Since $body$(Q) is a regular $body$, h is then a $homomorphism$ from $body$(Q) to $Body(\overline{Pick(C, (B', prov'))})$.

   On the other hand, according to Proposition 1.3.86, $B_{sq(C)} = Pick(C, (B_U, Prov_U))$. Let $B^{\mathcal{C}}_{sq(C)}$ be the result of a terminating Standard Chase sequence on $B_{sq(C)}$ with $\mathcal{C}$. By Theorem 1.3.84 it follows that $\overline{B^{\mathcal{C}}_{sq(C)}}$ and $Body(\overline{Pick(C, (B', prov'))})$ are homomorphically equivalent. In particular, there exists a $homomorphism$ h_1 from $Body(\overline{Pick(C, (B', prov'))})$ to $\overline{B^{\mathcal{C}}_{sq(C)}}$.

   But then $h_1 \circ h$ is a $homomorphism$ from $body$(Q) to $\overline{B^{\mathcal{C}}_{sq(C)}}$. By Proposition 1.3.85 it then follows directly that $sq(C)$ is a reformulation of Q.

2. Let sq be a reformulation of Q. Then there exists a conjunct $C' \in rf(F)$ such that $C' \subseteq C(sq)$.

   Indeed, let $B^{\mathcal{C}}_{sq}$ be the result of a terminating Standard Chase sequence with $\mathcal{C}$ over B_{sq}. It follows by Proposition 1.3.85 that there exists a $homomorphism$ h from $body$(Q) to $\overline{B^{\mathcal{C}}_{sq}}$.

   On the other hand, by Proposition 1.3.86, $B_{sq} = Pick(C(sq), (B_U, Prov_U))$. Then according to Theorem 1.3.84, $Body(\overline{Pick(C(sq), (B', prov'))})$ and $\overline{B^{\mathcal{C}}_{sq}}$ are homomorphically equivalent. In particular, there exists a $homomorphism$ h_1 from $\overline{B^{\mathcal{C}}_{sq}}$ to $\overline{Pick(C(sq), (B', prov'))}$.

   But then $h_2 = h_1 \circ h$ is a $homomorphism$ from $body(Q)$ to $\overline{Pick(C(sq), (B', prov'))}$. Then according to Lemma 1.3.78, h_2 is a $homomorphism$ from $body$(Q) to $(\overline{B'}, \overline{Prov}')$, such that $\overline{Prov}'(h_2(body(Q)) \prec C(sq)$. But by definition of F, $\overline{Prov}'(h_2(body(Q)) \subseteq F$. It follows from the subsumption that there exists $C' \in F$ s.t. $C' \subseteq C(sq)$. But by definition of the reduced form, for every $C' \in F$, there exists $C \in rf(F)$ such that $C \subseteq C'$.

Using the two points above, we will show that a subquery sq of U is a minimal reformulation of Q iff $C(sq) \in rf(F)$:

Indeed, let sq be a minimal reformulation. By point (2) above it follows that there exists $C' \in rf(F)$ such that $C' \subseteq C(sq)$. But by point (1) above it follows that $sq' = sq(C')$ is also a reformulation of Q. Since $sq(C') \subseteq sq(C)$, sq' is a subquery of sq. But sq is minimal so it must be that $sq' = sq$, therefore $C' = C(sq)$ and therefore $C(sq) \in rf(F)$.

Reversely, let $C \in rf(F)$ be a conjunct of $rf(F)$. By point (1) above it follows that $sq(C)$ is a reformulation of Q (since $rf(F) \subseteq F$). If $sq(C)$ is not minimal, that there exists $sq' \subset sq(C)$ s.t. sq' is a reformulation. But then by point (2) above there must exist $C' \in rf(F)$ s.t. $C' \subseteq C(sq')$. This further implies that that $C' \subset C$, which contradicts the definition of the reduced form of a formula. Therefore, $sq(C)$ must be minimal, which concludes our proof. □

1.4 Implementation

To evaluate our reformulation algorithm, we have set up a proof-of-concept implementation of $Prov_{C\&B}$. We present below some of the key techniques and optimizations employed:

Chase step as query evaluation. A Standard Chase step searches for homomorphic matches of the premise of constraint C against the closed version of a *body* B. The search for homomorphic matches of the premise can be modelled as running C's premise C_{prem}(viewed as a query) against $\overline{B}$ (viewed as a symbolic database known in the literature as the canonical instance [1]). We then compile C_{prem} to a query plan based on relational algebra operators, and we run it over an internal representation of $\overline{B}$ using its canonical instance. This technique can be further adapted to sk_bodies and the cs_chase, where the canonical database corresponds to $Body(\overline{B})$. We can further extend such technique for matching the conclusion of C in the case of the Standard Chase or the cs_chase with sk_EGDs. For the cs_chase with sk_TGDs on the other hand, such matching is greatly simplified by the fact that, as shown, a possible match of the conclusion is completely determined by the constructive terms of the images of distinguished premise terms.

Standard query optimizations. Modelling chase steps as query evaluation problems allows us to apply standard query optimization techniques borrowed from the relational query optimization literature. Our implementation includes among others pushing selections and (duplicate-eliminating) projections into the joins.

Efficient in-memory query processing. In contrast to general DBMSs that need to account for large datasets that may not fit in main memory, the chase operates on instances that start from a single query body and are small enough to fit in main memory. This observation allowed us to implement the chase engine as an in-memory query processor. To speed up query processing, we opted for in-memory hash-based implementations of the relational algebra operators (joins and projections).

Bottom-up query evaluation. In a naive chase engine, one would try to apply every constraint each time the instance changes. However, some constraints would not be applicable. To reduce the number of constraints we try to apply, our query processor works in a bottom-up fashion. Whenever a new tuple is added to a relation, it is being pushed up the query trees that scan this relation. Thus, for every change in the underlying instance only those queries that might be affected are evaluated.

Incremental query evaluation. A chase sequence involves evaluating repeatedly the same set of query plans obtained from compiling the constraints. Moreover, these queries are evaluated over evolutions of the *same* instance. The effect of each chase step is to evolve the instance by adding only a few new tuples at a time (these tuples correspond to the atoms constructed by the step). The majority of the instance is unaffected by the step. This creates the opportunity to accelerate chasing by employing incremental query evaluation. Instead of evaluating each query from scratch, we keep its query plan (together with the populated hash tables) in memory and whenever new tuples are added to the evolving instance, we push them to the affected plans.

Efficient maintenance of equalities. Chasing involves adding equalities between the values present in an instance. Moreover, it involves checking whether two values are equal. To allow efficient querying and updating of equalities, we employ a union-find data structure as in [60].

Our Provenance-Aware Chase implementation uses the design choices listed above, together with handling provenance formulae storage and operations. We have in particular optimized provenance enriching pa_chase steps, so as to *propagate* provenance changes without reevaluating the constraints. As a side-effect of our reformulation work, the pa_chase implementation further delivers a minimal-why-provenance-tracking processor for conjunctive queries (generalized to support invention of values using Skolem functions).

1.5 Experiments

We evaluated our $Prov_{C\&B}$ implementation in a recreation (and extension) of the setting described in [60] for query rewriting using materialized views and integrity constraints. We chose this setting because we believe it is practically relevant, it allows apples-to-apples comparison with the $C\&B$, and because its design was parameterized so as to allow scaling to the point of stress-testing any complete reformulation algorithm by forcing a combinatorial explosion of the existing minimal rewritings.

Chain-of-stars schemas and queries ([60]) The parameterized setting starts from the following basic building block. Consider the query Q given below, which joins relations $R_1(K, A_1, A_2, F)$, $R_2(K, A_1, A_2)$ with S_{ij} (A_i, B) $(1 \leq i, j \leq 2)$. Figure 1.1 depicts Q's join graph, in which the nodes represent the query variables and the edges represent equijoins between them.

Q: select $s_{11}.B$, $s_{12}.B$, $s_{21}.B$, $s_{22}.B$
from $R_1\ r_1$, $S_{11}\ s_{11}$, $S_{12}\ s_{12}$, $R_2\ r_2$, $S_{21}\ s_{21}$, $S_{22}\ s_{22}$
where $r_1.F = r_2.K$ and $r_1.A_1 = s_{11}.A_1$ and $r_1.A_2 = s_{12}.A_2$
and $r_2.A_1 = s_{21}.A_1$ and $r_2.A_2 = s_{22}.A_2$

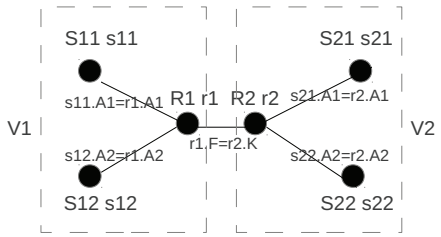


Figure 1.1: Join graph for a two-star query

One can think of the tables S_{ij} as modeling offered choices in two distinct domains, such as educational and recreational, grouped by several categories. The tables S_{11} and S_{12} could correspond to the lectures and workshops categories, while S_{21} and S_{22} could hold the sports and movies categories respectively. Categories span a range of subcategories (such as action movies), expressed by the A_j attributes, such that in every subcategory there are potentially many offered choices (the B attributes).

On the other hand, the tables R_1 and R_2 correspond to individual "preference profiles" in the respective domains, such that each profile selects, for a given category, either a specific subcategory or no preference (null). The K attributes are unique profile identifiers, thus primary keys. The join of R_1 and R_2 constructs global profiles for a group, with $R_1.F$ being a foreign key referencing K in R_2. Think of R_2 tuples as describing profiles of a "person" entity, while R_1 tuples describe profiles of a "student" entity, with the key-foreign key join implementing the "isA" relationship.

Towards identifying correlations of offered choices across domains, Q finds sets of choices that represent all categories and that co-occur within the global preference profile of some individual.

Assume the existence of materialized views $V_i(K, B_1, B_2)$ $(1 \leq i \leq 2)$, where each V_i joins R_i with S_{i1} and S_{i2} and retrieves the B attributes from S_{i1} and S_{i2} together with the key K of R_i :

V_i: select $r.K$, $s_1.B$, $s_2.B$
from $R_i\ r$, $S_{i1}\ s_1$, $S_{i2}\ s_2$
where $r.A_1 = s_1.A_1$ and $r.A_2 = s_2.A_2$

Assuming that only a small fraction of the individuals expresses preferences for all categories, the extent of the materialized views is expected to be relatively small, all the more so when considering that the same offering may appear in several subcategories, for instance action movies and comedies (recall our convention that all queries have an implicit DISTINCT keyword).

Since these views discard, for each domain, the unmatching profiles, we would expect them to be quite useful in speeding up Q's execution. It is easy to see that the join of R_2, S_{21}, and S_{22} can be replaced by a scan over V_2:

Q_1: select $s_{11}.B$, $s_{12}.B$, $v_2.B_1$, $v_2.B_2$
from $R_1\ r_1$, $S_{11}\ s_{11}$, $S_{12}\ s_{12}$, $V_2\ v_2$
where $r_1.F = v_2.K$ and $r_1.A_1 = s_{11}.A_1$ and $r_1.A_2 = s_{12}.A_2$

However, the join of R_1, S_{11}, and S_{12} cannot be blindly replaced by a scan over V_1, since Q_2, the obvious candidate for a rewriting of Q using both V_1 and V_2 is not equivalent to Q in the absence of additional semantic information.

Q_2: select $v_1.B_1$, $v_1.B_2$, $v_2.B_1$, $v_2.B_2$
from $R_1\ r_1$, $V_1\ v_1$, $V_2\ v_2$
where $r_1.K = v_1.K$ and $r_1.F = v_2.K$

The reason is that V_1 does not contain the F attribute of R_1, and there is no guarantee that joining the latter with V_1 will recover the correct values of F. On the other hand, *if we know that K is a key in R_1*, then Q_2 is guaranteed to be equivalent to Q, being therefore an additional (and likely better) plan. V_1 is usable for rewriting Q only by exploiting the key constraint.

Consider now a slightly more complicated version of the above configuration. The query graph is shaped like a chain of 2 stars, star i having R_i for its hub and S_{ij} for its corners ($1 \leq i \leq 2$, $1 \leq j \leq 3$). The attributes selected in the output are the B attributes of all corners S_{ij}. Assume the existence of materialized views $V_{il}(K, B_1, B_2)$ ($1 \leq i \leq 2$, $1 \leq l \leq 2$), where each V_{il} joins the hub of star i (R_i) with two of its consecutive corners (S_{il} and $S_{i(l+1)}$). Each V_{il} selects the B attributes of the corner relations it joins, as well as the K attribute of R_i, as depicted in Figure 1.2.

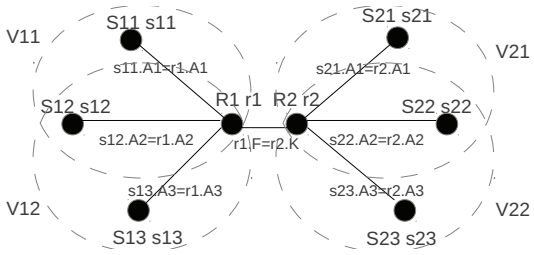


Figure 1.2: Chain-of stars configuration with 3 corners

Notice that in this setting *all* views require the key constraint to be usable in rewriting.

The chain-of-star configuration generalizes to chains of H stars with C corners each, such that for each star there are $C - 1$ views, each joining the hub with two consecutive corners. As soon as C is greater than 2, the key constraint on the hub table is a prerequisite for the usability of every view involving that hub. Note that the chain-of-star schema shape is inspired by such patterns as star and snowflake schemas, which are well-represented in practice [45].

Platform All experiments were run on an Intel(R) Core(TM) i7-2760QM CPU @ 2.40GHz with 8GB of memory.

Experiment 1: Is complete search worthwhile? We investigated whether the potential overhead induced by running the complete search for rewritings given by $Prov_{C\&B}$ is justified by the speedup achieved over the execution of the original query without using $Prov_{C\&B}$. To assess this speedup, we performed a suite of comparative experiments with a well-known and widely used commercial DBMS. We compared two alternatives: (a) feeding the original query "as is" to the DBMS, versus (b) feeding it the rewriting obtained by running $Prov_{C\&B}$ to enumerate all minimal rewritings using the views and integrity constraints, then picking among these one rewriting with the overall minimum number of joins (randomly selecting one if several exist). Note that we are placing the $Prov_{C\&B}$ on top of the DBMS, which gives a lower bound to the speedup potential achievable by a tighter integration with the DBMS's optimizer.

For the purpose of our experiments, we constructed a chain-of-stars schema, with 5 stars and 5 corners/star, for a total of 30 tables, 20 materialized views, and 5 key constraints. This schema was then extended with an additional 25 tables and 25 foreign key constraints to a total of 55

tables, as described in Experiment 2. The table contents obey the following statistics, which are compatible with the real-life interpretation of our scenario:

- the cardinality of the views V_{ij} is 10% of that of the tables R_i

- we ensure 5% selectivity for the joins between R_i and S_{ij}

Over this schema we ran chain-of-stars queries of various complexity levels, up to a maximum number of 20 joins (the DBMS was timing out too frequently after that), thus leading to the following configurations (our figures refer to them):

#stars	2	3	4	5	2	3	4	5	2	3	4	2	3
#corners	2	2	2	2	3	3	3	3	4	4	4	5	5
#joins	5	8	11	14	7	11	15	19	9	14	19	11	18

For each query, we measured the following elapsed times:

Q_{exec}: the time taken by the DBMS to execute (optimize then run) the original query.

RW_{find}: the time taken by our $Prov_{C\&B}$ implementation to find all minimal rewritings and choose one with the fewest joins.

RW_{exec}: the time taken by the DBMS to execute (optimize then run) this rewriting.

We populated each table in our schema with 5K tuples, generated randomly according to our selectivity parameters (the DBMS automatically created indexes on all key attributes). We enabled the use of materialized views in the optimizer. We set a timeout of 15 minutes (900 seconds) for query execution times. We used the recommended optimization level, which comes preset out of the box[7].

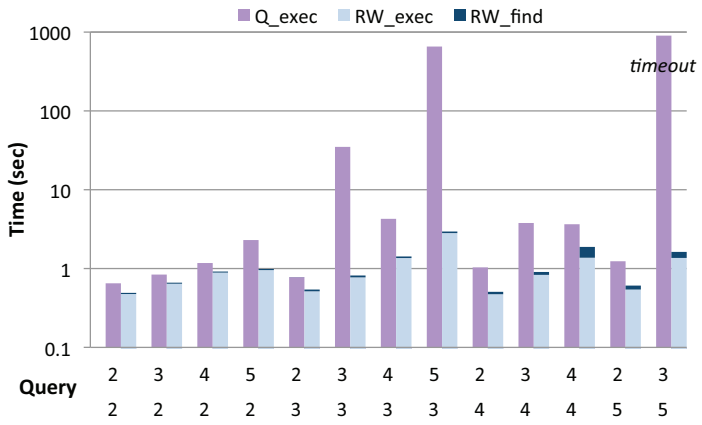


Figure 1.3: Elapsed times on one database instance

[7]For fairness we considered all optimization levels. Out of a total of 7, only 4 consider materialized views, and two of these are designed for ultra-specialized queries, spending so much time in optimizing our queries that the optimization time vastly dominates execution time. The remaining two view-aware levels, call L_r the recommended one and L_a the alternative, are similar except that L_a uses a greedy algorithm while L_r uses dynamic programming for join reordering. The speedups for $Prov_{C\&B}$ we observed under L_a are generally even higher than the ones we observed under L_r (we omit them for space reasons).

Figure 1.3 presents the measured values for Q_{exec}, RW_{find} and RW_{exec}, for each of the tested queries. Query (s, c) refers to the configuration with s stars of c corners each. In the graph, s appears above c. Times RW_{find} and RW_{exec} are shown stacked into the same bar, as this is the total time taken when we interpose $Prov_{C\&B}$ before calling the DBMS. Notice that, for all the queries, RW_{find} is a very small fraction of Q_{exec}, which in turn stays larger than the sum of RW_{find} and RW_{exec} even for the smallest query. Also notice that the speedup yielded by $Prov_{C\&B}$ can reach one, and even two orders of magnitude.

The reason we never observe parity between Q_{exec} and RW_{exec} is that the minimum-join rewriting found using $Prov_{C\&B}$ uses views extensively (as explained for query Q_2 above), while the DBMS *fails to detect that views are relevant whenever doing so requires exploiting the key constraint.* The DBMS-provided explanation of the plan choice states that the views were considered but rejected because of the missing foreign key attribute. The only exception when a view is indeed used is for the last star of 2-cornered star queries ((2,2) through (5,2)) because this view is relevant even without the key constraint (recall the discussion for Q_1 above).

The drop in the measured Q_{exec} time from (3,3) to (4,3) is interesting: it is due to the fact that we imposed no restriction on the join between two consecutive stars, other than it being a foreign key join (this is consistent with the targeted real-life scenario interpretation described above). Generally, it may happen that adding a new star to the query actually "filters out" a lot of results. If the filtering join is performed early enough by the DBMS, its small intermediate result can propagate its impact to any cross-star intermediate chain of joins. This is exactly what occurred in this case (as an inspection of the plan explanation confirmed).

This observation called for better accounting of the execution time variations due to the actual data. We therefore repeated the experiment over 10 different randomly populated database instances obeying the same table-size and selectivity criteria. For each database instance and query, we computed the *speedup factor*, defined as *speedup factor* $= \frac{Q_{exec}}{RW_{find}+RW_{exec}}$.

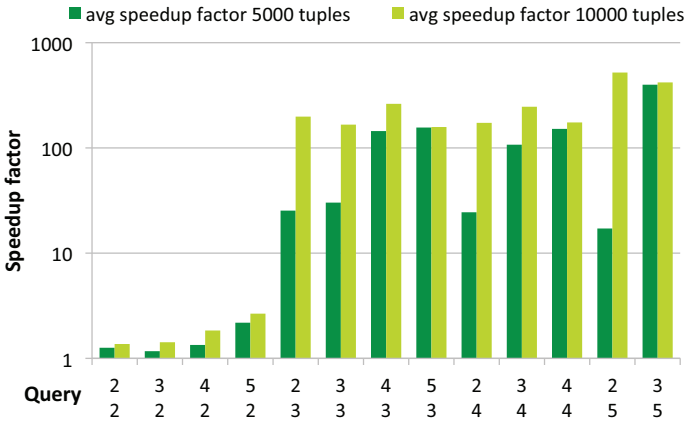


Figure 1.4: Average speedup factors on 10 database instances

Figure 1.4 presents, for each query, the speedup factors averaged over the set of 10 database instances, providing a more robust view of the advantage of the rewritings according to the query complexity. Notice that the measurements in Figure 1.3 were not a lucky fluke, being quite typical (the speedups are in many cases below average). Note that the values for queries

(5,3), (4,4) and (3,5) are only lower bounds for the speedup, because these queries time out on several databases.

We remind that, at 5K tuples per table, the database instances are rather small. We repeated the experiments for larger tables of 10K tuples each (timeouts while measuring Q_{exec} prevented us from pushing the experiment any further). Observe that the average speedups increase, a trend we expect to continue with increasing data size. Timeouts are once again responsible for the seemingly marginal increases for queries (5,3), (4,4) and (3,5), since the figure only reports a lower bound for the average speedup.

In conclusion, on small-sized queries the performance of the DBMS's processing engine, coupled with the ability of its optimizer to use views (for the last star of 2-cornered configurations, for which the key constraint is not needed) leads to fast query execution. Although for every database instance and every query we ran, the measured speedup factors are higher than 1 (calling $Prov_{C\&B}$ still results in a speed-up), on the small-sized queries they are less pronounced. On the other hand, as the query complexity increases, the time Q_{exec} dramatically increases, up to the order of minutes even on relatively small instances, and the speedup factors become significantly more substantial as using the views makes increasing difference. As Figure 1.4 shows, on more complex queries the view-based plans gain an advantage of one and even two orders of magnitude (and often more, but this is masked by the timeouts when measuring Q_{exec}).

Experiment 2: Performance of the $Prov_{C\&B}$ implementation We further analyzed the standalone performance of our implementation. In our evaluation, we also studied the behaviour of our algorithm beyond key constraints. We extended the chain-of-stars schema to also incorporate foreign keys, by adding the tables $T_{ij}(B,C)$ such that (see Figure 1.5):

- $S_{ij}.B$ is a foreign key referencing $T_{ij}.B$

- the views output the same attributes, but also contain a join with the T tables.

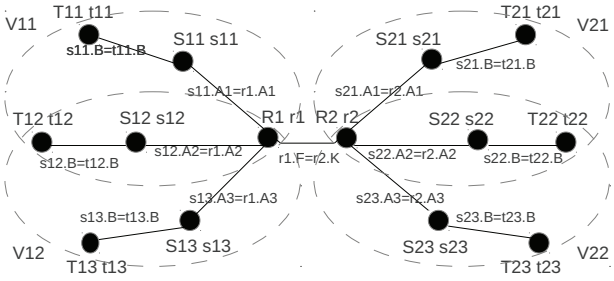


Figure 1.5: Extended chain-of-stars configuration

The chain-of-stars queries over the new schema, hereafter called the "extended chain-of-stars schema", stay the same. The views however are now recognizable as relevant for rewriting only when exploiting *both keys and foreign keys constraints*. The resulting view-based rewritings are identical to the ones in Experiment 1. The $Prov_{C\&B}$ implementation continues to find them, while the DBMS continues to miss them. This time, the DBMS misses even the views it used to find for the 2-corner case. Their detection now involves reasoning about the foreign keys, which

is evidently incomplete in the DBMS. RW_{exec} does not change, while Q_{exec} increases for the 2-corner queries, leading to increased speedups.

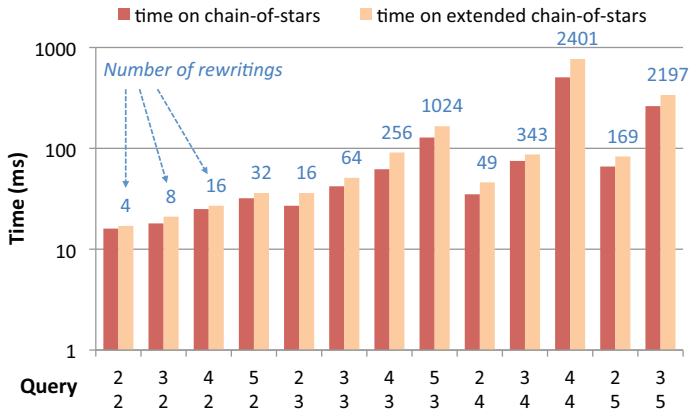


Figure 1.6: Rewrite computation times (RW_{find})

We omit their values, focusing instead on reporting RW_{find} in Figure 1.6, which shows average times over the 10 runs (rewriting is unaffected by the database instance, and the measured times are virtually identical). The graph shows rewrite computation times on the two schemas (as expected the foreign keys cause more work, but the difference is not substantial). The two schema types are chosen such that a large number of minimal rewritings is available in the large configurations, to enable a stress-test of our implementation as it pursues all rewritings. In Figure 1.6, we annotate each query with the number of minimal rewritings it admits (all of whom are found) shown as a bar label. On both schemas, our implementation exhibits sub-second running times. This is true even for configurations with over 2000 minimal rewritings, e.g. (4,4). Note that the rewrite computation times represent a very small fraction of the query execution times reported in Experiment 1.

Experiment 3: Savings over the $C\&B$ Recall that the original motivation for the design of $Prov_{C\&B}$ was to save the chase sequences launched by the $C\&B$ algorithm during the backchase phase. We quantify (a lower bound for) these savings here. For both the chain-of-stars schema and its extended version, the $C\&B$ backchase (called the Full Backchase in [60]) will chase at least each actual minimal rewriting to determine its equivalence to the original query. The $Prov_{C\&B}$ saves at least all these chases, whose number is depicted in Figure 1.6 as bar labels. We note the exponential trend of savings as the number of hubs or corners increases.

1.6 Mininum-cost reformulations with $Prov_{C\&B}$

While the previous sections show how $Prov_{C\&B}$ allows finding *all* minimal reformulations of a query, it is often the case in practice that we are interested in the *minimum* reformulations according to a cost function.

A vast majority of the cost functions encountered in practice are *monotonic cost functions*

Definition 1.6.1 (Monotonic cost function)**.** *A cost function γ is said to be monotonic if for every query Q and every subquery Q' of Q, $\gamma(Q') \leq \gamma(Q)$.*

As already mentioned in the introductory section, monotonicity of a cost function has a strong consequence: it allows us to state that the *minimum-cost* reformulations of a query Q are always among the *minimal* ones:

Proposition 1.6.2. *Let Q be a query formulated against a source schema S and T a target schema. Let γ be a monotonic cost function and Q' a minimum-cost reformulation of Q against T according to γ.*

Then Q' is a minimal reformulation of Q.

The above proposition suggests a simple strategy for computing minimum reformulations with respect to a monotonic cost function γ. For a query Q formulated against a source schema S, a target schema T and a set of weakly acyclic constraints $\mathcal{C}$ over $S \cup T$:

1. compute $RW(Q, S, T, \mathcal{C}) = Prov_{C\&B}(Q, S, T, \mathcal{C})$, the minimal reformulations of Q.
2. return $\{argmin\, \gamma(rw_i), rw_i \in RW\}$.

Note that our strategy for choosing a reformulation in our experimental evaluation (Section 1.5) corresponds to the reasoning above, where the cost function is the number of joins in the query.

While provably correct according to Proposition 1.6.2 and the soundness and completeness of $Prov_{C\&B}$, the naive algorithm above involves however generating all minimal reformulations before choosing among them a minimum one. While in our experimental evaluation we have shown that the efficiency of $Prov_{C\&B}$ allows achieving very satisfactory performance by proceeding as above, we will show hereafter that we can further speedup computation when the aim is to find only minimum-cost reformulations.

Indeed, the properties of the pa_chase allow for a more refined approach: the *cost-based pruning* of the provenance formulae while chasing, thus significantly reducing the search space for minimum-cost reformulations. We will dedicate the rest of this section to describing a modified version of the $Prov_{C\&B}$ that includes such pruning, and to presenting its soundness and completeness guarantees.

1.6.1 Cost-based pruned Provenance-Aware Chase steps

Remember that in $Prov_{C\&B}$, the minimal reformulations of a query are represented by the provenance conjuncts in a formula. The conjuncts of this formula in turn are obtained by multiplying conjuncts in individual atom provenance adornments.

The first main observation concerning cost-based pruning is that a conjunct in an atom's adornment whose (corresponding subquery's) cost is larger than the minimum cost will never participate in a minimum-cost rewriting (this is a direct consequence of the monotonicity of the cost function).

Intuitively, one can thus simply "cut out" these conjuncts from the provenance formulae while running the pa_chase. Defining the cost of a conjunct to be the cost of the corresponding

subquery, we formalize such "cutting out" by means of a pruning function denoted by $Prune$, taking as input a threshold T and a provenance formula and yielding as output a provenance formula, as follows:

Definition 1.6.3. *Let $F = C_1 + \cdots + C_n$ a provenance formula, γ a cost function and T a quantity in the target domain of γ called a* threshold*. Then $Prune(T, F)$=$C_{i_1} + \cdots + C_{i_k} \subseteq F$, s.t. a conjunct C_{i_j} is in $Prune(T, F)$ iff $\gamma(C_{i_j}) <= T$.*

Pruned pa_chase steps. Using the function $Prune$ defined above, we hereafter introduce the notion of pruned pa_chase steps as follows:

Definition 1.6.4 (Pruned pa_chase step conditions of application)**.** *A pruned pa_chase step with $sk\_unit\_constraint$ C and threshold T on a* provenance-adorned sk_body *$(B, Prov)$ applies iff:*

1. *There exists a $homomorphism$ h from C_{prem} to $\overline{B}$ such that $Prune(T, \overline{Prov}(h(C_{prem}))) \neq \emptyset$*

2. *Either:*

   (a) *there exists no $homomorphism$ h' compatible with h from C_{concl} to $\overline{B}$, or*

   (b) *for any such h', $\overline{Prov}$(h'(C_{concl})) $\not\asymp Prune(T, \overline{Prov}(h(C_{prem})))$*

Definition 1.6.5 (Pruned pa_chase step application)**.** *Applying a pruned pa_chase step with $sk\_unit\_constraint$ C and threshold T on a* provenance-adorned sk_body *$(B, Prov)$, given $homomorphism$ h from C_{prem} to $\overline{B}$, results in a new* provenance-adorned sk_body *$(B', Prov') = Pruned\_Pa\_Chase\_Step\_Res((B, prov), C, h, T)$, $B' \supseteq B$, obtained in the exact same manner as for a regular pa_chase step, but employing P_{prem} = $Prune$(T,$\overline{Prov}$(h(C_{prem})))*.

1.6.2 Cost-based pruned $Prov_{C\&B}$

The second main observation that leads to the design of the modified version of $Prov_{C\&B}$ aimed towards finding minimum-cost reformulations, is that we can "incrementally" compute the provenance formula F giving the reformulations of a query Q, by *interlacing* computation of $homomorphisms$ from the $body$ of Q to the *provenance-adorned sk_body* with the pa_chase steps.

Then, if instead of regular pa_chase steps, we employ cost-based pruned pa_chase steps, such interlacing will allow us to *adjust the threshold* corresponding to the pruned pa_chase steps. We will thus combine cost-based pruned pa_chase steps and incremental reformulation computation. We present below the resulting algorithm, called PRUNED $Prov_{C\&B}$:

PRUNED $Prov_{C\&B}$ (Query Q, source schema S, target schema T, set of weakly acyclic constraints $\mathcal{C}$, monotonic cost function γ)

1 $B_U \leftarrow Universal\ Body(Q, \mathcal{C}, S, T)$
2 $Prov_U \leftarrow$ canonical adornment of B_U
3 $(B', Prov') \leftarrow (B_U, Prov_U)$
4 $RF \leftarrow$ GET_RW_FORM$(Q, (B', Prov'))$
5 $Th \leftarrow \min \gamma(C_i), C_i \in RF$
6 **while** there exists at least one $sk\_unit\_constraint\ C \in skunit(\mathcal{C})$ s.t. a pruned pa_chase step with C and Th applies on $(B', Prov')$
7 **do**
8 Pick $C \in skunit(\mathcal{C})$ s.t. a pruned pa_chase step with C and Th applies on $(B', Prov')$, with a $homomorphism\ h$ from C_{prem} to $(\overline{B'}, \overline{Prov'})$
9 $(B', Prov') \leftarrow Pruned\_Pa\_Chase\_Step\_Res((B', Prov'), C, h, Th)$
10 $RF \leftarrow$ GET_RW_FORM$(Q, (B', Prov'))$
11 $Th \leftarrow \min \gamma(C_i), C_i \in RF$
12 **for** $rw \in argmin\ \gamma(C_i), C_i \in RF$
13 **do**
14 **return** $sq(rw)$

where the brick GET_RW_FORM encompasses the computation of $homomorphisms$ and their formula, as follows:

GET_RW_FORM (Query Q, *provenance-adorned sk_body* (B', $Prov'$))

1 $\mathcal{H} \leftarrow \{h,\ h\ is\ a\ homomorphism\ from\ body(Q)\ to\ (\overline{B'}, \overline{Prov'})\}$
2 $F \leftarrow \sum_{h \in \mathcal{H}} \overline{Prov'}(h(body(Q))$
3 **return** $rf(F)$

We claim that the above algorithm is sound and complete for computing minimum-cost reformulations for monotonic cost functions, as follows:

Theorem 1.6.6. *Let Q be a SFW query formulated over a source schema S, T a target schema and $\mathcal{C}$ a set of weakly acyclic constraints. Let γ be a monotonic cost function.*

Then the algorithm PRUNED $Prov_{C\&B}$ *is sound and complete, that is, it returns* all and precisely *the minimum-cost reformulations of Q against T given $\mathcal{C}$ and γ.*

Proof. Let $(B_0 = B, Prov_0 = Prov)$, $(B_1, Prov_1)$, ..., be the sequence of pruned pa_chase steps in PRUNED $Prov_{C\&B}$, and $Th_0, Th_1, \ldots$ be the corresponding thresholds.

We will exhibit a "lock-step" pa_chase sequence $(B'_0 = B, Prov'_0 = Prov)$, $(B'_1, Prov'_1)$, ..., such it respects the following properties:

1. there exists an isomorphism H_i from $\overline{B_i}$ to $\overline{B'_i}$
2. for every atom a in $\overline{B_i}$, $\overline{Prov_i}(a) \subseteq \overline{Prov'_i}(H_i(a))$
3. moreover, if a conjunct cj is in $\overline{Prov'_i}(a)$ and $\gamma(cj) \leq Th_i$ then $cj \in \overline{Prov_i}(H_i^{-1}(a))$.

We first note that if the properties above hold for $(B_i, Prov_i)$ and $(B'_i, Prov'_i)$ then they also hold for $(\overline{B_i}, \overline{Prov_i})$ and $(\overline{B'_i}, \overline{Prov'_i})$ (this is a direct consequence of isomorphisms and provenance of closed versions). We further note that $Th_{i-1} \geq Th_i$.

We construct the sequence $(B'_i, Prov'_i)$, by letting $(B'_i, Prov'_i) =$ $Pa\_Chase\_Step\_Res((B'_{i-1}, Prov'_{i-1}), C_i, H_i \circ h_i)$, where C_i is the $sk\_unit\_constraint$ corresponding to the pruned pa_chase step $i-1 \longrightarrow i$ and h_i is the $homomorphism$ from $C_{i\,prem}$ to $\overline{B_{i-1}}$.

The properties above (existence of an isomorphism, inclusion and preservation of conjuncts with cost lower or equal to Th_i) obviously hold for $i = 0$, since the *provenance-adorned sk_bodies* are identical. We show inductively that if they hold for $i-1$ then they hold for i (and thus the pa_chase sequence is also correctly defined).

Indeed, we first show that if the pruned pa_chase step applies, then the corresponding pa_chase step applies. For an atom creation step, given the isomorphism H_{i-1}, the property obviously holds, furthermore allowing us to deduce the isomorphism H_i (this is a direct consequence of the cs_chase and Lemma 1.3.61).

Furthermore, by the induction hypothesis it is easy to show that $\overline{Prov}_{i-1}(h_i(C_{prem})) \subseteq \overline{Prov}'_{i-1}(H_{i-1} \circ h_i(C_{prem}))$ (this is a simple consequence of multiplying pairwise included boolean formulae). Since pruning only removes some conjuncts from $\overline{Prov}_{i-1}(h_i(C_{prem}))$, for the newly introduced atoms, which by definition are adorned with the provenance of the image of the premise, the inclusion property is further respected. Also, note that for every conjunct cj in $\overline{Prov}'_{i-1}(H_{i-1} \circ h_i(C_{prem}))$ such that $\gamma(cj) \leq Th_i$, due to the monotonicity of the cost function and the definition of the provenance of a set of atoms, the following holds: for every atom a in $H_{i-1} \circ h_i(C_{prem})$ there exists $cj_a \in \overline{Prov}'(a)$ such that $\gamma(cj_a) \leq Th_i$, and cj is equal to the product of all cj_a.

But then since $Th_{i-1} \geq Th_i$, according to the induction hypothesis cj_a is also in $\overline{Prov}_{i-1}(H_{i-1}^{-1}(a))$, therefore cj is also in $\overline{Prov}_{i-1}(h_i(C_{prem}))$. On the other hand, , by definition of the pruned pa_chase step, it follows that cj is also in $Prune(Th_{i-i}, \overline{Prov}_{i-1}(h_i(C_{prem})))$, thus the preservation of conjuncts is further respected for the newly introduced atoms.

For a provenance enriching step, we will rely on the same type of reasoning as above by further noting that if h'_i is the pruned pa_chase step compatible $homomorphism$ then $H_{i-1} \circ h'_i$ is a pa_chase step compatible $homomorphism$ for the pa_chase sequence, and if $Prune(Th_{i-1}, \overline{Prov}_{i-1}(h_i(C_{prem}))) \neq \phi$ and $\overline{Prov}_{i-1}(h'_i(C_{concl})) \not\prec$ $Prune(Th_{i-1}, \overline{Prov}_{i-1}(h_i(C_{prem})))$ then it is equally the case that $\overline{Prov}'_{i-1}(H_{i-1} \circ h'_i(C_{concl}))$ $\not\prec \overline{Prov}'_{i-1}(H_{i-1} \circ h_i(C_{prem}))$. Indeed, since the pruned pa_chase step applies, it means that conjuncts with cost lower than Th_{i-1} exist in the provenance of the image of the premise for the pruned pa_chase step. According to the properties linking the two sequences, it follows that all such conjuncts also exist for the pa_chase sequence. Assuming the pa_chase step does not apply, they could only be subsumed by conjuncts with cost lower than Th_{i-1} in the provenance of the image of the conclusion $\overline{Prov}'_{i-1}(H_{i-1} \circ h'_i(C_{concl}))$. Accordingly, all those subsuming conjuncts would have to exist in $\overline{Prov}_{i-1}(h'_i(C_{concl}))$, thus the pruned pa_chase step would not apply.

It further straightforward to show that if the pa_chase sequence terminates, then the pruned pa_chase sequence also terminates. Indeed, we have shown that if a pruned pa_chase step ap-

plies then the corresponding pa_chase step (through the isomorphism exhibited) must apply.

Given the properties exhibited by the two lock-step sequences, we further note that, given a query Q, they can be extended to $homomorphisms$ from $body$(Q) to the respective outputs of chase steps. We can then claim the following:

- for every conjunct cj in GET_RW_FORM$(Q, (B_i, Prov_i))$, cj is also in GET_RW_FORM$(Q, (B'_i, Prov'_i))$. Furthermore, if cj' is a conjunct in GET_RW_FORM$(Q, (B'_i, Prov'_i))$ such that $\gamma(cj') \leq Th_i$, then cj' is also in GET_RW_FORM$(Q, (B_i, Prov_i))$.

The reasoning is very similar to that applying to individual chase steps, and easily extended to reduced forms.

Let us now suppose that the pruned pa_chase sequence has terminated after a number k of steps. Let Th_k be the corresponding threshold as computed by PRUNED $Prov_{C\&B}$. Then $\min \gamma(cj), cj \in$ GET_RW_FORM$(Q, (B_k, Prov_k))$ = Th_k.

But by the above properties we can conclude that
{argmin γ(cj), cj in GET_RW_FORM$(Q, (B'_k, Prov'_k))$ } =
{argmin γ(cj), cj in GET_RW_FORM$(Q, (B_k, Prov_k))$ }
and
$\min \gamma(cj), cj \in$ GET_RW_FORM$(Q, (B'_k, Prov'_k))$ =
$\min \gamma(cj), cj \in$ GET_RW_FORM$(Q, (B_k, Prov_k))$ = Th_k.

To conclude, we further show that for any continuation of the pa_chase sequence with some pa_chase steps $k+1, k+2, \ldots$, no minimum-cost reformulations are added. That is, we show that for every $i \geq k$,
{argmin γ(cj), cj in GET_RW_FORM$(Q, (B'_i, Prov'_i))$ } =
{argmin γ(cj), cj in GET_RW_FORM$(Q, (B'_k, Prov'_k))$ }.

Indeed, it is enough to show that for every $i > k$, and every conjunct cj in GET_RW_FORM$(Q, (B'_i, Prov'_i))$ - GET_RW_FORM$(Q, (B'_k, Prov'_k))$, $\gamma(cj) > Th_k$.

To prove the above, we show by induction on the pa_chase steps that the following hold:

1. for every $homomorphism$ h from C_{prem} to $(\overline{B'_i}, \overline{Prov'_i})$, if a pa_chase step with h applies, then for every conjunct $cj' \in \overline{Prov'_i}(h(C_{prem}))$, $\gamma(cj') > Th_k$.
2. every atom a in $B'_i - B'_k$ is such that for every conjunct cj' in $Prov'_i(a)$, $\gamma(cj') > Th_k$
3. every atom a in B'_k is such that for every conjunct cj' in $Prov'_i(a) - Prov'_k(a)$, $\gamma(cj') > Th_k$

We first note that if the first property in the list above holds for i, then the second and third will necessarily hold for $i+1$, by definition of the pa_chase step.

The first property obviously holds for the pa_chase step $k \longrightarrow k+1$, otherwise we can easily show that the pruned pa_chase sequence would not have terminated. Then the second

and third properties hold for $(B'_{k+1}, Prov'_{k+1})$. In turn, for the subsequent pa_chase step on $(B'_{k+1}, Prov'_{k+1})$, if the mapping of the premise only uses atoms from $\overline{B_k}$, then we can apply the same reasoning (i.e. non-termination of the pruned pa_chase sequence), coupled to the third property above. Else, we use the second property above. In the two cases, we can thus re-infer the first property on $(B'_{k+1}, Prov'_{k+1})$ and in the same inductive manner on all $(B'_i, Prov'_i)$.

Putting together the above results,and using the fact that *any* pa_chase sequence can be applied in $Prov_{C\&B}$, it follows that computing the minimum-cost reformulations on $(B'_k, Prov'_k)$ ensures that *all and precisely* the minimum-cost reformulations are found:

$\{argmin\ \gamma(rw), rw \in Prov_{C\&B}(Q, S, T, \mathcal{C})\}$ =

$\{sq(cj)\}, cj \in \{argmin\ \gamma(cj'), cj' \in$ GET_RW_FORM$(Q, (B'_k, Prov'_k))\}$

On the other hand, we have shown above that

{argmin γ(cj), $cj \in$ GET_RW_FORM$(Q, (B'_k, Prov'_k))$ } =

{argmin γ(cj), $cj \in$ GET_RW_FORM$(Q, (B_k, Prov_k))$ }

But PRUNED $Prov_{C\&B}(Q, S, T, \mathcal{C})$ = $\{sq(cj')\}$,

$cj' \in \{argmin\ \gamma(cj), cj \in$ GET_RW_FORM$(Q, (B_k, Prov_k))\}$.

It follows that PRUNED $Prov_{C\&B}(Q, S, T, \mathcal{C})$ = $\{argmin\ \gamma(rw), rw \in Prov_{C\&B}(Q, S, T, \mathcal{C})\}$, which concludes our soundness and completeness proof. □

1.6.3 Initial experimental evaluation

To test the benefits of employing PRUNED $Prov_{C\&B}$ for minimum-cost reformulations, we revisit our experimental setting by choosing as a cost function the same cost function as in Section 1.5, that is, the number of joins of the rewriting. We compare the following:

1. the time spent by employing $Prov_{C\&B}$ for finding all minimal reformulations + the time (in reality, negligible) of selecting all minimum-cost ones, versus

2. the time spent by employing PRUNED $Prov_{C\&B}$ for finding all minumum-cost reformulations.

We employ the same chain-of-stars configurations as in previous experiments (recall that a query is defined by its number of hubs and corners).

Figure 1.7 shows the time measured for the two strategies employed (for accuracy of comparison, the graph is no longer shown on a logarithmic scale). Note that we can obtain up to six times speedup with PRUNED $Prov_{C\&B}$. Note further that the speedup importantly increases with the complexity of the query, and that PRUNED $Prov_{C\&B}$ exhibits extremely high performance on all the considered configurations.

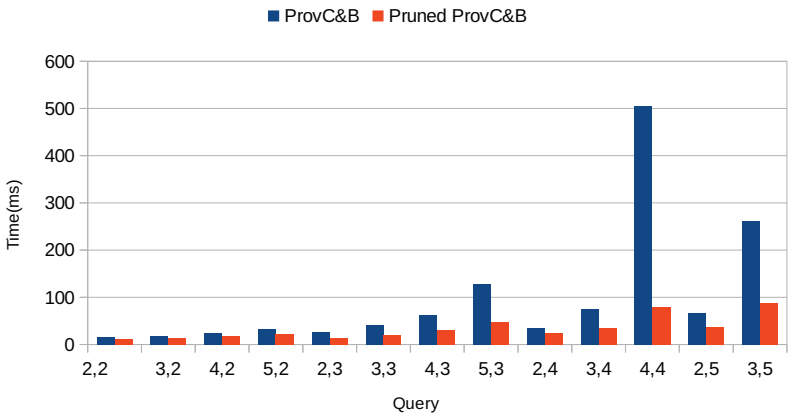


Figure 1.7: Comparison of $Prov_{C\&B}$ and PRUNED $Prov_{C\&B}$

1.7 Related work

The problem of query reformulation includes view-based rewriting as particular case. This problem is fundamental to many classic data management tasks, including query optimization using materialized views, data security and integration. It represents a fruitful research area and has been treated in depth for relational databases, for a wide spectrum of model assumptions, from those pertaining to the language of queries and views [46, 4, 3, 21, 64], to going from set semantics to bag or mixed bag-set semantics (see [23] and the references therein), to adding limited access patterns for the views [35, 55], or to using potentially infinite sets of views [47]. In the context of information integration, view-based rewriting has been extended also to finding not-necessarily equivalent (but maximally-contained) rewritings (see [40] and references therein).

The first complete view-based rewriting algorithm for the SQL fragment considered in this paper, in the absence of integrity constraints, was given in [46], where the problem was shown NP-complete. In practice, this leads to either deterministic exponential-time implementations, or to algorithms that rely on view-matching heuristics (e.g., [36, 44, 71, 8]), which are potentially more efficient but may fail to identify some rewriting opportunities. Such heuristic-based approaches may also assume an integrated process within the DBMS's optimizer module, comparing the cost of the found rewritings to that of plans without views.

In the presence of constraints, the $C\&B$ [27], discussed at length in this paper, is the only complete algorithm we are aware of in this setting. As emphasized however, its completeness fails to achieve practical performance because of the important number of subquery chases launched during the backchase phase. The idea of speeding up the $C\&B$ by using provenance information was first mentioned in [26], becoming this thesis' topic due the complexity and the theoretical depth of the problem, which we reveal in our work.

In the Provenance-Aware Chase, the provenance bookkeeping exploits the analogy between chase step application and query evaluation, with the provenance annotations coinciding with the *minimal why-provenance* flavor introduced for query evaluation in [14], and corresponding to a particular case of a provenance semiring [38]. Recently, we have witnessed revived interest in the

chase, with studies such as [52, 50, 51] focusing in particular on more permissive conditions than weak acyclicity that can guarantee termination. The Skolemization procedure on the constraints, that we use in the Conservative Chase to reach the purpose of sound provenance bookkeeping, is also used in the Semi-Oblivious chase [50, 51], to attain specific termination purposes.

The original $C\&B$ algorithm has been extended in follow-up work to apply *beyond conjunctive queries* (see [28] for a survey of these extensions). The extensions allow disjunction/union [31], nested correlated query blocks, grouping, aggregation, user-defined functions, and show a uniform way to incorporate any additional language primitives by treating them as user-defined functions with black-box semantics [57, 70]. Moreover, extensions support additional data models, such as object-oriented, complex-valued [27] and XML [29, 31, 70]. Not surprisingly, once the supported language features sufficient expressive power even checking equivalence becomes undecidable, so all hope is dashed for a complete reformulation algorithm. However, all existing $C\&B$ extensions still guarantee soundness, i.e. only equivalent reformulations are reported. They also guarantee to continue finding, within a larger query, all reformulations of the query's fragments that correspond to some language with complete $C\&B$ (or extension thereof). All $C\&B$ extensions transfer directly to the $Prov_{C\&B}$ algorithm as they are all reduced to the original $C\&B$, relying solely on the input-output behaviour of the $C\&B$ (shared by the $Prov_{C\&B}$) and not on its internal working.

Recent work [11] on accommodating non-terminating chase sequences argues trading completeness in favour of producing *low-cost reformulations*, emphasizing their practical interest. While it is beyond the scope of this work, we note that such behaviour can be achieved with the cost-based pruned version of $Prov_{C\&B}$ presented in Section 1.6, by *merging* the chase and backchase phase in a single provenance-aware, cost-based pruned sequence.

Chapter 2

A theoretical and practical approach to finding XPath rewritings with a single-level of intersection of multiple views

We revisit in this chapter the work of Cautis, Deutsch and Onose, presented in [16] and detailed in [56], on the problem of finding XPath rewritings with a single level of intersection of multiple views.

XPath [20] is the standard for navigational queries over XML data and it is widely used, either directly, or as part of more complex languages (such as XQuery [13]). Early studies such as [68, 48, 65, 69] have considered the problem of rewriting XPath queries by navigating inside a *single* view's output, which is the only possible kind of rewritings supported when in the materialized views the original node identities are lost. The industrial trend towards enhancing XPath queries with the ability to expose node identifiers and exploit them using intersection, supported by such systems as [7], has led to the adoption of intersection as a first-class primitive of the XPath standard, starting from XPath 2.0 [12] and through the XPath 3.0 standard [62]. The ability of persisting node identifiers in materialized views provides in turn the opportunity of rewriting for a much larger set of queries than those rewritable using a single view, by employing the intersection of the results of several materialized views.

The work presented in [16] and detailed in [56] investigates the intersection-aware rewriting problem, focusing on rewritings with a single level of intersection of multiple views: that is, rewritings where navigation is performed in the views, then intersection occurs, then potential additional navigation may be applied. The authors characterize the complexity of this problem and provide a sound and complete algorithm for its resolution. In the light of the proven hardness results, they further present a sound rule-based procedure and its usage for inferring a sound algorithm for the rewriting problem, also describing conditions for this sound algorithm to become complete.

The main motivation of the contributions presented in this chapter is that of investigating and achieving practical performance for the rewriting setting presented above. To this purpose, we

refine the rule-based procedure to ensure its polynomial complexity, improve the completeness of the resulting rewrite procedure, and present a range of optimizations that are necessary for obtaining practically-relevant running time. We further provide a complete implementation of the rewriting algorithms, employing our refinements and optimizations, as well as a thorough experimental evaluation thereof, showing the performance and the practical benefits of the refined and optimized polynomial rewriting techniques. As a side effect of reviewing the work in [16] and [56], we also contribute in enriching the analysis of the rewriting problem by showing, structuring and clarifying its connections with the problem of deciding the equivalence between a query expressed as a DAG pattern and a query expressed as a tree pattern, and to the problem of union-freeness (finding any tree pattern equivalent to a DAG pattern query).

The remainder of this chapter is organized as follows: we start by recalling, in Section 2.1, the rewriting problem and the general sound and complete rewriting algorithm described in [16]. We dedicate Section 2.2 to showing the strong link between the rewriting problem and the DAG-tree equivalence and union-freeness problems. In Section 2.3 we present our refinement of the rule-based algorithm, and show its usage to infer sound polynomial algorithms for the three related problems described in Section 2.2. We recall and refine conditions for the completeness of these algorithms in Section 2.4. We describe our complete implementation of the rewriting procedures and the many optimizations it comprises in Section 2.5, and its extensive experimental evaluation in Section 2.6. We present related work in Section 2.7.

2.1 View-based rewritings

We dedicate this section to recalling the contents of [16] and [56], defining the rewriting problem and describing a general sound and complete algorithm for its resolution. In order to ensure readability, we also recall the necessary preliminary notions, and restructure and refine the material from [16] and [56] to improve the clarity of further theoretical developments.

In the following, according to the approach from [16], an XML document D is considered as an unranked, unordered rooted tree t, modelled by a set of edges $\text{EDGES}(t)$, a set of nodes $\text{NODES}(t)$, a distinguished root node $\text{ROOT}(t)$ and a labelling function λ_t, assigning to each node a label from an infinite alphabet Σ, such that $\lambda_t(\text{ROOT}(t)) = "doc("D")"$. Every node n in the tree has a text value $\mathit{text}(n)$, possibly empty.

2.1.1 *XP* queries and tree patterns

We recall in this subsection the subset of XPath considered in [16], denoted by *XP*. *XP* comprises queries with child / and descendant // navigation, without wildcards, whose grammar can be represented as follows:

$$
\begin{array}{rcl}
\mathit{apath} & ::= & \mathit{doc}(\text{“}name\text{”})/\mathit{rpath} \mid \mathit{doc}(\text{“}name\text{”})//\mathit{rpath} \\
\mathit{rpath} & ::= & \mathit{step} \mid \mathit{rpath}/\mathit{rpath} \mid \mathit{rpath}//\mathit{rpath} \\
\mathit{step} & ::= & \mathit{label}\ \mathit{pred} \\
\mathit{pred} & ::= & \epsilon \mid [\mathit{rpath}] \mid [\mathit{rpath} = C] \mid [.//\mathit{rpath}] \mid [.//\mathit{rpath} = C] \mid \mathit{pred}\ \mathit{pred}
\end{array}
$$

Expressions in *XP* are produced from the symbol *apath* and they correspond to *absolute paths*, that is, queries expressed starting from the document root. The *rpath* symbol generates *relative path* expressions, i.e. encoding navigation relative to a given document context. The sub-expressions inside brackets are called *predicates*. C terminals stand for text constants, while "name" is a placeholder for an actual document name.

As noted in [16], *XP* queries are further representable by an adaptation of the unary *tree patterns* [53]:

Definition 2.1.1. *A* tree pattern p *is a non empty rooted tree, with a set of nodes* $\text{NODES}(p)$ *labelled with symbols from* Σ *by a labelling function* λ_p*, a distinguished node called the* output node $\text{OUT}(p)$*, and two types of edges:* child edges*, labelled by* $/$ *and* descendant edges*, labelled by* $//$*. The root of* p *is denoted* $\text{ROOT}(p)$*. Every node* n *in* p *has a test of equality test*(n) *that is either the empty word* ϵ*, or a constant* C*. If* n *is on a path between* $\text{ROOT}(p)$ *and* $\text{OUT}(p)$*, then test*(n) *is* ϵ*.*

For a given *XP* expression q, *pattern*(q) denotes the associated tree pattern p and *xpath*$(p) = q$ the reverse transformation.

2.1.2 $XP^{\cap-simple}$, $XP^{\cap}$, DAG patterns

We present in this subsection two extensions of *XP* with respect to intersection. The first language considered, called $XP^{\cap-simple}$, is obtained by adding the following rule to the grammar of *XP*:

$$cpath \quad ::= \quad apath \mid cpath \cap apath$$

Expressions in $XP^{\cap-simple}$ are produced by the symbol *cpath* which defines a *single level of intersection* of *XP* expressions, e.g. doc("v_1")/image $\cap$ doc("v_2")/image. Further enriching the grammar of $XP^{\cap-simple}$ with the following rule:

$$ipath \quad ::= \quad cpath \mid (cpath)/rpath \mid (cpath)//rpath$$

provides the language $XP^{\cap}$, which is the focus of the rewriting study in [16]. Note that *ipath* adds to the single-level intersection an *rpath* expression, thus allowing additional (relative) navigation from the nodes in the intersection result, e.g. (doc("v_1")/image $\cap$ doc("v_2")/image)/file.

The $XP^{\cap-simple}$ language is not presented in a standalone manner in [16] or [56], being instead implicitly considered as a sublanguage of $XP^{\cap}$. We provide its standalone definition above as we consider this distinction necessary for the clarity of the developments hereafter.

By $XP^{\cap-simple}$ and $XP^{\cap}$ expressions over a set of documents D we denote those that use only *apath* expressions that navigate inside the documents D (i.e. starting with doc("name") where $name \in D$). For a fragment $\mathcal{L} \subseteq XP$, by $XP^{\cap-simple}(\mathcal{L})$ we will denote $XP^{\cap-simple}$ queries that use only *apath* expressions from $\mathcal{L}$.

While *XP* queries can be represented by tree patterns, queries in $XP^{\cap-simple}$ and $XP^{\cap}$ are representable [16] by the more general *DAG patterns*:

Definition 2.1.2. *A* DAG pattern d *is a directed acyclic graph, with a set of nodes* $\text{NODES}(d)$ *labeled with symbols from* Σ *by a labeling function* λ_d*, a distinguished node called the* output node $\text{OUT}(d)$*, and two types of edges:* child edges*, labeled by* $/$ *and* descendant edges*, labeled by* $//$*.* d *has to satisfy the property that any* $n \in \text{NODES}(d)$ *is accessible via a path starting from a special node* $\text{ROOT}(d)$*. In addition, all the nodes that are not on a path from* $\text{ROOT}(d)$ *to* $\text{OUT}(d)$ *(called* predicate nodes*) have only one incoming edge. Every node* n *in* d *has a test of equality test*(n) *that is either the empty word* ϵ*, or a constant* C*. If* n *is on a path between* $\text{ROOT}(d)$ *and* $\text{OUT}(d)$*, then test*(n) *is always* ϵ*.*

For a query q in $XP^{\cap-simple}$, the associated DAG pattern can be constructed as follows:

1. for every *apath* (*XP* path with no $\cap$), $\texttt{dag}$(*apath*) is the tree pattern corresponding to the *apath* .
2. $\texttt{dag}(p_1 \cap p_2)$ is obtained from $\texttt{dag}(p_1)$ and $\texttt{dag}(p_2)$ as follows: (i) provided p_1 and p_2 are not empty and there are no labeling conflicts between their root and output nodes, by coalescing $\text{ROOT}(\texttt{dag}(p_1))$ with $\text{ROOT}(\texttt{dag}(p_2))$ and $\text{OUT}(\texttt{dag}(p_1))$ with $\text{OUT}(\texttt{dag}(p_2))$ respectively, (ii) otherwise, as the empty pattern.

Figure 2.1(a) gives an example of a DAG pattern corresponding to a query in $XP^{\cap-simple}$ which intersects the queries doc("L")//paper//section[theorem]//image and doc("L")/lib/paper//section//figure[caption//label]/image). For the depicted DAG, $\text{ROOT}(d)$ is the $doc(L)$ node and $\text{OUT}(d)$ is the *image* node indicated by a square. Note that in practice an $XP^{\cap-simple}$ expression is *representable by a non-empty DAG* iff the *apath* expressions are *over the same document* and furthermore their end labels coincide.

For queries in $XP^{\cap}$, $\texttt{dag}(x/rpath)$ and $\texttt{dag}(x//rpath)$ are obtained as follows: (i) for non-empty x, by appending the pattern corresponding to *rpath* to $\text{OUT}(\texttt{dag}(x))$ with a /- and a //-edge respectively, (ii) as x, if x is the empty pattern.

By a slight abuse of terminology, we will use for DAGs corresponding to queries in $XP^{\cap-simple}$ the denomination *DAGs in* $XP^{\cap-simple}$, and similarly refer to DAGs in $XP^{\cap}$. In the following, unless explicitly stated otherwise, the notion of *pattern* refers to both DAG and tree patterns. We recall hereafter several concepts related to patterns.

Main branches and main branch nodes. By the main branch nodes of a pattern d, $\text{MBN}(d)$, we denote the set of nodes found on paths starting with $\text{ROOT}(d)$ and ending with $\text{OUT}(d)$. We refer paths between $\text{ROOT}(d)$ and $\text{OUT}(d)$ as *main branches* of d. By definition, a tree pattern p has a unique main branch, which we denote by $\text{MB}(p)$.

Predicate subtrees. We call *predicate subtree* of a pattern p any subtree of p rooted at a non-main branch node. A */-predicate* (resp. *//-predicate*) is a predicate subtree connected by a /-edge (resp. //-edge) to a main branch node. As further specialization, by a */-subpredicate* st we denote a predicate subtree whose root is connected by a /-path to the main branch node to which st is associated. By a *//-subpredicate st* we denote a predicate subtree whose root is connected by a //-edge to a /-path p that comes from the main branch node n to which st is associated (as in $n[\ldots[.//st]]$). p is called the *incoming /-path* of st and can be empty, when st is a //-predicate.

Subpatterns. We further denote by a *subpattern* of a pattern d any pattern that could be obtained from a pattern d by removing some nodes and edges. For a pattern d and node $n \in \text{MBN}(d)$, by $\text{SP}_d(n)$ we denote the subpattern rooted at n in d.

Prefixes. A prefix p of a tree pattern q is any tree pattern with $\text{ROOT}(p) = \text{ROOT}(q)$, $m = \text{MB}(p)$ a subpath of $\text{MB}(q)$ and having all the predicates attached to the nodes of m in q. A *lossless prefix* p of a tree pattern q is any tree pattern obtained from q by setting the output node to some other main branch node (i.e., an ancestor of $\text{OUT}(q)$). Note that this means that the rest of the main branch becomes a side branch, hence a predicate.

Tokens (/-patterns). A *token*, also called */-pattern*, is a tree pattern that has only child (/) edges in the main branch. Tokens provide a means of reasoning about tree patterns in general. Indeed, the main branch of a tree pattern p can be partitioned in tokens by its sub-sequences separated by //-edges. We can thus see any tree pattern p as a sequence of tokens $p = t_1//t_2//\ldots//t_k$. We call t_1, the token starting with $\text{ROOT}(p)$, the *root token* of p. The token t_k, which ends by $\text{OUT}(p)$, is called the *result* or *output token* of p. The other tokens are denoted *intermediary tokens*, and by the *intermediary part* of a tree pattern we denote the sequence of intermediary tokens.

2.1.3 Pattern satisfiability, containment and equivalence

We summarize in this subsection concepts and results previously presented in literature, regarding the satisfiability, containment and equivalence of (tree or DAG) patterns. We start by recalling the notions of satisfiability, containment and equivalence, as well as those of mappings between patterns:

Definition 2.1.3. *A pattern d is* satisfiable *if it is non-empty and there exists a tree t over Σ into which d has an embedding (i.e., there exists a model with non-empty results).*

Definition 2.1.4. *A pattern d_1 is* contained *in another pattern d_2 iff for any input tree t, $d_1(t) \subseteq d_2(t)$. We write this shortly as $d_1 \sqsubseteq d_2$. We say that d_1 is* equivalent *to d_2, and write $d_1 \equiv d_2$, iff $d_1(t) = d_2(t)$ for any input tree t.*

Definition 2.1.5. *A* mapping *between two patterns d_1 and d_2 is a function $h : \text{NODES}(d_1) \to \text{NODES}(d_2)$ that satisfies the following properties:*

1. *for any $n \in \text{MBN}(d_1)$, $h(n) \in \text{MBN}(d_2)$*
2. *for any $n \in \text{NODES}(d_1)$, $\lambda_{d_2}(h(n)) = \lambda_{d_1}(n)$*
3. *for any /-edge (n_1, n_2) in d_1, $(h(n_1), h(n_2))$ is a /-edge in d_2*
4. *for any //-edge (n_1, n_2) in d_1, there is a path from $h(n_1)$ to $h(n_2)$ in d_2*
5. *for any $n \in \text{NODES}(d_1)$, if test$(n) = C$ then test$(h(n)) = C$*

A *root-mapping* is a mapping that further satisfies the following: $h(\text{ROOT}(d_1)) = \text{ROOT}(d_2)$ A *containment mapping* is a root-mapping h such that further $h(\text{OUT}(d_1)) = \text{OUT}(d_2)$. An *isomorphism* between d_1 and d_2 is a bijective containment mapping from d_1 into d_2 whose inverse is also a containment mapping, from d_2 into d_1. We recall hereafter several well known results from previous literature (e.g., [48]) linking containment mappings, containment and equivalence:

Lemma 2.1.6. *If there is a containment mapping from a pattern d_1 to a pattern d_2 then $d_2 \sqsubseteq d_1$.*

Lemma 2.1.7. *A tree pattern p is contained in a DAG pattern d iff there is a containment mapping from d to p.*

Lemma 2.1.8. *Containment and equivalence for two tree patterns p_1 and p_2 can be evaluated in PTIME.*

2.1.4 Interleavings

We recall in this subsection a central notion for characterizing DAG patterns, namely their *interleavings*. Interleavings are intuitively "foldings", or "zippings" of a DAG pattern into a tree, formally defined as follows:

Definition 2.1.9 (Interleaving)**.** *An* interleaving *of a pattern d is any tree pattern p_i produced as follows:*

1. *choose a string i of Σ symbols alternating with either / or // (we call such string a* code *[10]) and a total onto function f_i that maps $\text{MBN}(d)$ into Σ-positions of i such that:*
   - *(a) f_i is label preserving*
   - *(b) for any /-edge (n_1, n_2) in d s.t. $n_1 \in \text{MBN}(d)$ and $n_2 \in \text{MBN}(d)$, the code i is of the form $\ldots f_i(n_1)/f_i(n_2)\ldots$,*
   - *(c) for any //-edge (n_1, n_2) in d s.t. $n_1 \in \text{MBN}(d)$ and $n_2 \in \text{MBN}(d)$, the code i is of the form $\ldots f_i(n_1)\ldots f_i(n_2)\ldots$.*
2. *build the smallest tree pattern p_i such that:*
   - *(a) i is a code for the main branch $\text{MB}(p_i)$ (i corresponds to $\text{MB}(p_i)$'s string representation))*
   - *(b) for any $n \in \text{MBN}(d)$ and its image n' in p_i (via f_i), if a predicate subtree st appears below n then a copy of st appears below n', connected by same kind of edge.*

Two nodes n_1, n_2 from $\text{MBN}(d)$ are said to be collapsed *(or* coalesced*) if $f_i(n_1) = f_i(n_2)$, with f_i as above. The tree patterns p_i thus obtained are called* interleavings *of d and we denote their set by* interleave(d)*.*

Figure 2.1(c) shows an interleaving of the DAG pattern in Figure 2.1(a).

An immediate observation is that if d is satisfiable, then the set *interleave(d)* is non-empty. By definition, there is always a containment mapping from a satisfiable pattern into each of its interleavings. Then, by Lemma 2.1.6, a pattern will always contain its interleavings. Moreover [37, 10], it also holds that:

Lemma 2.1.10. *Any DAG pattern is equivalent to the union of its interleavings.*

Note that the set of interleavings of a DAG pattern d can be exponentially larger than d. Indeed, it was shown [10] that a DAG pattern may only be translatable into a union of exponentially many tree patterns.

2.1.5 Union-freeness, dominant interleavings and DAG-tree equivalence

We recall in this subsection a central property of DAG patterns, their *union-freeness* [16]. A DAG pattern d is *union-free* iff there exists a tree pattern p such that d and p are equivalent. We further define *the problem of union-freeness* in its decision and functional versions, as follows:

- *decision version:* Given a DAG pattern d, decide whether d is union-free.
- *functional version:* Given a DAG pattern d, exhibit a tree pattern equivalent to d iff such pattern exists. We will call such pattern a *tree equivalent of* d.

Note that the notion of union-freeness encompasses that of satisfiability. Note also that the functional version of the union-freeness problem encompasses its decision version. In the following, when referring to the union-freeness problem, we will always designate its functional version.

Based on union-freeness, we can straightforwardly characterize the equivalence between a DAG and a tree as follows:

Proposition 2.1.11. *A DAG pattern d is equivalent to a tree pattern p iff d is union-free and for p' a tree equivalent of d, p' is equivalent to p.*

We further recall the very strong link that exists between union-freeness and interleavings. Indeed, by Lemma 2.1.10, a DAG pattern is equivalent to the union of its interleavings. Furthermore, the following also hold[1]:

Proposition 2.1.12. *Let $p = \cup_i p_i$ and $q = \cup_j q_j$ be two finite unions of tree patterns. Then $p \sqsubseteq q$ iff $\forall i, \exists j$ s.t. $p_i \sqsubseteq q_j$.*

Proposition 2.1.13. *If a tree pattern is equivalent to a union of tree patterns, then it is equivalent to a member of the union.*

Given a DAG pattern d, by the *normal form of* d we denote the equivalent formulation of d as the union of incomparable interleavings with respect to containment. It follows that the union-freeness of a DAG can be characterized as follows:

Lemma 2.1.14. *A DAG pattern is* union-free *iff its normal form contains a single interleaving. Such interleaving is then a solution for the problem of union-freeness (a tree equivalent of the given DAG).*

In other words, the above result states that a DAG pattern is union-free iff it has an interleaving that contains all the others. We will call such interleaving a *dominant interleaving*.

[1] reminiscence of similar results from relational database theory, on comparing conjunctive queries with unions of conjunctive queries

2.1.6 The view-based rewriting problem for $XP^{\cap}$

We recall in this subsection the problem of query rewriting using views with *rewrite plans in* $XP^{\cap}$, as described in [16], as well as its complexity, as claimed in [16] and proven in [56].

We consider views defined by queries over a document D. For a view v, by $\bar{v}$ we denote the query defining it. We further assume that for each view v, the result of executing $\bar{v}$ over the document D is materialized in a corresponding *view document* v, such that the topmost element is labelled with $doc(\text{“}v\text{”})$ and its children subtrees are Id-preserving copies of the subtrees of D, rooted at the nodes selected by $\bar{v}$ over D. Given a set of views $\mathcal{V}$ defined by *XP* queries over a document D, by $D_{\mathcal{V}}$ we denote the set of view documents $\{v | v \in \mathcal{V}\}$, containing the materialized results of executing the corresponding queries.

Rewrite plans in $XP^{\cap}$ .A *rewrite plan in* $XP^{\cap}$ over $D_{\mathcal{V}}$ is a query $r \in XP^{\cap}$ over the view documents $D_{\mathcal{V}}$, According to the definition of the $XP^{\cap}$ language, a rewrite plan r is then of the form $\bigcap_{i,j} u_{ij}$, $(\bigcap_{i,j} u_{ij})/rpath$ or $(\bigcap_{i,j} u_{ij})//rpath$, with u_{ij} of the form $doc(\text{“}v_j\text{”})/p_i$.

Unfolding rewrite plans. Given a rewrite plan r, its unfolding, denoted $unfold(r)$, is the $XP^{\cap}$ query obtained by replacing in r each $doc(\text{“}v\text{”})$ label with $\bar{v}$, the *XP* query defining v. Note that for a rewrite plan in $XP^{\cap}$, $unfold(r)$ will always represent a query in $XP^{\cap}$ over *a single document* D, which represents the document the views have been defined over.

View-based rewriting problem for $XP^{\cap}$. Relying on the above concepts, the view-based rewriting problem for $XP^{\cap}$ is defined as follows: for q an *XP* query over a document D and $\mathcal{V}$ a finite set of views over D, find a rewrite plan r in $XP^{\cap}$ over $D_{\mathcal{V}}$ such that $unfold(r)$ and q are equivalent. Such plan is then called a *rewriting*.

Example 2.1.15. *Given the following query* q *and views* v_1 *and* v_2*:*

$$q : \textit{doc(“L”)/lib/paper//section[theorem]//figure[caption//label]/image/file}$$

$$v_1 : \textit{doc(“L”)//paper//section[theorem]//image}$$

$$v_2 : \textit{doc(“L”)/lib/paper//section//figure[caption//label]/image}$$

the query r : *(doc(“*v_1*”)/image* $\cap$ *doc(“*v_2*”)/image)/file is a rewriting of* q *in* $XP^{\cap}$.

We recall hereafter the complexity result regarding the rewriting problem as stated in [16] and proven in [56]:

Theorem 2.1.16. *The rewriting problem for queries and views from XP and plans in* $XP^{\cap}$ *is coNP-complete.*

2.1.7 A sound and complete rewriting algorithm

We recall in this subsection the sound and complete rewriting algorithm REWRITE presented in [16]. As in [16], the compensate function generalizes the concatenation operation from [68], by copying extra navigation from the query into the rewrite plan. For a query $r \in XP^{\cap}$ and a tree pattern p, $\text{compensate}(r, p, n)$ returns the query obtained by deleting the first symbol from $x = xpath(\text{SP}_p(n))$ and concatenating the rest to r. For instance, the result of compensating r = a/b with x = b[c][d]/e at the b-node is the concatenation of a/b and [c][d]/e, i.e. a/b[c][d]/e. We present below the flow of the REWRITE algorithm, in which for clarity we have further emphasized its sub-algorithm, BUILDINITREWRITECANDIDATE.

REWRITE$(q, \mathcal{V})$

1 **for** p a lossless prefix of *pattern*(q)
2 **do**
3 $r \leftarrow$ BUILDINITREWRITECANDIDATE$(p, \mathcal{V})$
4 $d \leftarrow$ *pattern*(*unfold*(r))
5 **if** $d \equiv p$
6 **then return** compensate$(r, q,$ OUT$(p))$
7 **return fail**

BUILDINITREWRITECANDIDATE$(p, \mathcal{V})$

1 $\mathcal{V}' \leftarrow \phi$
2 **for** $v \in \mathcal{V}$, h a root-mapping of *pattern*$(\bar{v})$ into p
3 **do**
4 $b \leftarrow h($OUT(*pattern*$(\bar{v})))$
5 $\mathcal{V}' \leftarrow \mathcal{V}' \cup$ compensate (doc("v")/$\lambda_p(b), p, b)$
6 $r \leftarrow \left(\bigcap_{v_j \in \mathcal{V}'} v_j\right)$
7 **return** r

We also recall the soundness and completeness guarantees of REWRITE :

Theorem 2.1.17. REWRITE *is sound, that is, if it returns an XP$^{\cap}$ expression, then this expression is a rewriting for q.*

Furthermore, REWRITE *is complete, that is, if there exists a rewriting in XP$^{\cap}$ for q, then* REWRITE *will return a result.*

Note that the completeness concept as defined by [16] is close to the corresponding decision problem: indeed, in order to be complete, an algorithm solving the rewrite problem must return a non-empty result as soon as a rewriting exists.

Note also that in the version of REWRITE provided in [16] and [56], the equivalence test between d and p appears as a *containment* test (i.e. if $d \sqsubseteq p$), due to the strategy of construction for d, ensuring that the opposite containment always holds. We state explicitly the equivalence test in the above in order to clarify and structure the results in the following.

2.1.8 Interesting *XP* fragments

We dedicate this subsection to recalling the two *XP* fragments further considered in the theoretical developments of [16] and [56]:

The fragment *XP*$_{es}$. This fragment comprises queries p called *extended skeletons*, in which the usage of //-subpredicates is limited as follows: for any main branch node $n \neq$ OUT(p) and //-subpredicate st of n, there is no mapping (in either direction) between the code of the incoming /-path of st and the one of the /-path following n in the main branch (where the empty code is assumed to map in any other code). E.g., patterns a[b//c]/d//e or a[b//c//d]/e//d are extended skeletons, while a[b//c]/b//d, a[b//c]//d, a[.//b]/c//d or a[.//b]//c are not.

Observe that the above definition imposes no restrictions on predicates of the output node. This relaxation was not present in [16]'s definition of extended skeletons but it is easy to show that it does not affect any of the results that were obtained with the more restrictive definition. This is because there is only one choice for ordering the output nodes in interleavings of an $XP^{\cap-simple}$ intersection: they are collapsed into one output node.

The fragment $XP_{//}$. This fragment is an extension of XP_{es}, where //-predicates attached to main branch nodes are allowed and the usage of //-subpredicates therein is further freely allowed.

2.2 Rewritings, equivalence and union-freeness

We dedicate this section to showing the tight link that exists between the rewriting problem, the problem of deciding the equivalence between a DAG pattern and a tree pattern and the (functional version of the) union-freeness of a DAG pattern. In doing do, we provide a clear and structured framework for the intuitions and results presented in [16] and detailed in [56]. We will follow this framework throughout following sections, in order to structure and clarify the presentation of the results from [16], and to further enhance their applicability.

2.2.1 Rewritings and the DAG-tree equivalence

Remember that REWRITE uses as a central brick the equivalence test between a DAG in $XP^{\cap-simple}$ (corresponding to the unfolding of the rewrite candidate for a prefix) and a tree (the prefix). Note also that the number of such tests corresponds to the number of prefixes, and is thus linear in the size of the main branch of the input query.

As a consequence of the proven soundness and completeness of REWRITE it follows directly that the following holds:

Lemma 2.2.1. *The rewriting problem for $XP^{\cap}$ rewrite plans has a polynomial-time reduction to the problem of deciding equivalence between a DAG pattern in $XP^{\cap-simple}$ and a tree pattern.*

In view of the complexity results of the previous section and the above reduction we can then characterize the complexity of the DAG-tree equivalence problem as follows:

Theorem 2.2.2. *The problem of testing equivalence between an $XP^{\cap-simple}$ DAG pattern d and a tree pattern p in XP is coNP-complete.*

Proof. The lower-bound follows directly from Theorem 2.1.16. To show that the problem is in coNP, we note that a non-deterministic algorithm that decides $d \not\equiv p$ can guess a tree equivalent for d and check that $u \not\equiv p$, which can be done in PTIME according to Lemma 2.1.8. □

2.2.2 DAG-tree equivalence and union-freeness

Remember that by Proposition 2.1.11 the DAG-tree equivalence reduces to the (functional version of) the problem of union-freeness: A DAG pattern d is equivalent to a tree pattern p iff d is union-free and for p' a tree equivalent of d, p' is equivalent to p.

By Lemma 2.1.8 it further holds that the equivalence test for two tree patterns is PTIME. The following then holds:

Lemma 2.2.3. *The DAG-tree equivalence for $XP^{\cap-simple}$ DAGs and XP trees has a polynomial-time reduction to the problem of union-freeness for $XP^{\cap-simple}$ DAGs.*

Then, the hardness result of Theorem 2.2.2 transfers to the complexity of the union freeness problem as follows:

Theorem 2.2.4. *The (functional version of the) union freeness problem for a DAG pattern in $XP^{\cap-simple}$ is coNP-hard.*

Naively solving the problem of union-freeness. How does one go about solving the union-freeness problem? Remember that by Lemma 2.1.14, a dominant interleaving (or the non-existence thereof) provides a solution for union-freeness. A naive approach would then be the following algorithm:

DOMINANT_INTERLEAVING(d)
1 generate all interleavings of d
2 check whether they reduce by containment to a single interleaving
3 if so, output the dominant interleaving, else output $\emptyset$

It is easy to show that DOMINANT_INTERLEAVING is sound and complete for solving the union-freeness problem. Given the reductions stated above, we can further use DOMINANT_INTERLEAVING for solving the DAG-tree equivalence problem, as well as the rewriting problem.

2.3 A rule-based algorithm for directly constructing the dominant interleaving

While the algorithm DOMINANT_INTERLEAVING presented in the previous section is sound and complete, it may not be the best choice in terms of computational effort. Indeed, we have already emphasized the fact that the number of interleavings of a DAG d may be exponentially larger than d. A central development in [16] concerns the design of an algorithm for *directly constructing the dominant interleaving*, without going through the steps of generating all interleavings and checking their reduction by containment.

This approach consists in a series of transformations of the input DAG such that in the end "it becomes" its dominant interleaving. Each of these transformations is formalized as the application of a *transformation rule*, bringing the DAG one step closer to its dominant interleaving, if such interleaving exists.

We present hereafter a refinement of the rule-based algorithm in [16]. This refinement, for which we preserve the original name APPLY-RULES, is aimed towards achieving polynomial complexity and improving this algorithm's completeness, as we will show in the following.

2.3.1 Global flow of APPLY-RULES

We show below the global form of our refinement of the rule-based algorithm:

```
APPLY-RULES(d)
 1  d' = d
 2  if one of the patterns intersected in d is a /-pattern
 3     then RuleSet = R1, R2, R3, R4, R6, R7
 4     else RuleSet = R1, R2, R3, R4, R5, R6
 5  repeat
 6           while R1 applies on d'
 7              do
 8                 d' = apply R1 on d'
 9           if one Ri in RuleSet applies on d'
10             then d' = apply Ri on d';
11             else break;
12  return d'
```

Note that, compared to [16], this flow is modified in order to *ensure application of rule R1* after each of the other rules' application. Indeed, this application is necessary in order to ensure the uniqueness of /-paths between two nodes, which in turn is necessary for ensuring polynomial complexity for the individual rules.

Furthermore, the original statement of the rule-based algorithm presents 8 rules, which are not differentiated to account for the two cases above. In our refinement of the rule-based algorithm, we exhibit such a differentiation and we only employ 7 rules, as follows:

- Rules R1, R2, R3 and R4 stay the same as in [16].
- Rules R5 and R8 are further replaced by a new rule R7
- Accordingly, rule R6 in [16] becomes rule R5 in our refinement, and rule R7 in [16] becomes rule R6.

Rule R7 extends the combined effect of rules R5 and R8 in [16]. Its purpose is twofold: first, we show that its testing and application can be achieved in polynomial time, property that is not ensured by the previous rule R8; furthermore, together with the differentiation above, this rule ensures completeness of the rule-based algorithm for extended skeletons, as analyzed in Section 2.4.

2.3.2 The rewrite rules of APPLY-RULES

We list hereafter the 7 rules employed in APPLY-RULES. Following the approach in [16], each rule R1-R7 will be presented as a pair formed by a test condition, which checks if the rule is applicable (i.e. if the input DAG exhibits a required configuration), and a graphical description, which shows how the rule transforms the DAG. Each transformation either (i) collapses two main branch nodes n_1, n_2 into a new node $n_{1,2}$ (which inherits the predicate subtrees, incoming

and outgoing main branch edges), or (ii) removes some redundant main branch nodes and edges, or (iii) appends a new predicate subtree below an existing main branch node.

We use the graphical notation of [16]: linear paths corresponding to part of a main branch are designated by the letter p, nodes are designated by the letter n, the result of collapsing two nodes n_i, n_j is denoted $n_{i,j}$. Simple lines represent /-edges, double lines represent //-edges, simple dotted lines represent /-paths, and double dotted lines represent arbitrary paths (may have both / and //). We represent by a rhombus main branch paths that are not followed by any / (main branch) edge. Paths include their end points.

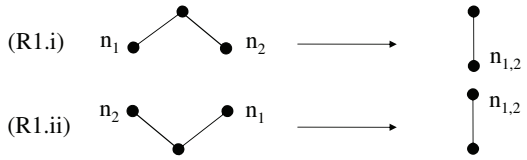
As in [16], the tree pattern containing just a main branch path p is referred to by p, and the tree pattern having p as main branch by $\text{TP}_d(p)$. We recall the definition of *immediate unsatisfiability* from [16]: a pattern d is *immediately unsatisfiable* if by applying to saturation rule R1 on it we reach a pattern in which either there are two /-paths of different lengths but with the same start and end node, or there is a node with two incoming /-edges λ_1/λ and λ_2/λ, such that $\lambda_1 \neq \lambda_2$. As in [16], two nodes n_1, n_2 are *collapsible* iff they have the same label and the DAG pattern $\mathit{collapse}_d(n_1, n_2)$ is not immediately unsatisfiable.

We also recall the notion of similar patterns in [16]:

Definition 2.3.1. *Two /-patterns p_1, p_2 are* similar *if (a) their main branches have the same code, and (b) both have root mappings into any pattern p_{12} built from p_1, p_2 as follows:*

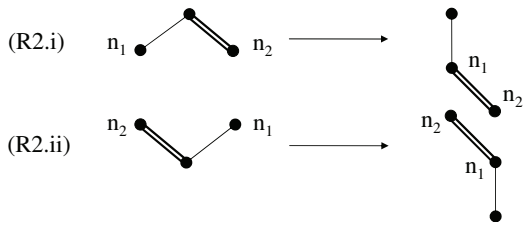
1. *choose a code i_{12} and a total onto function f_{12} that maps the nodes of $m_{12} = \text{MBN}(p_1) \cup \text{MBN}(p_2)$ into i_{12} such that:*
   - *(a) f_{12} preserves labels*
   - *(b) for any /-edge (n_1, n_2) in the main branch of p_1 or p_2, the code i_{12} contains $f_{12}(n_1)/f_{12}(n_2)$*
2. *build the minimal pattern p_{12} such that:*
   - *(a) i_{12} is a code for the main branch $\text{MB}(p_{12})$,*
   - *(b) for each node n in $\text{MBN}(p_1) \cup \text{MBN}(p_2)$ and its image n' in $\text{MB}(p_{12})$ (via f_{12}), if a predicate subtree st appears below n then a copy of st appears below n', connected by the same kind of edge.*

Rule R1. This rule triggers when $\lambda_d(n_1) = \lambda_d(n_2)$



Example 2.3.2. *The DAG pattern that would be obtained by intersecting some two tree patterns doc("L")/paper//... and doc("L")/paper/... would be subject to R1's application, with n_1 and n_2 being its two nodes labeled paper.*

Rule R2. This rule triggers if n_1 and n_2 are not collapsible and n_2 is not reachable from n_1 (resp. n_1 is not reachable from n_2, in the case of R2.ii).



Example 2.3.3. *Notice the application of rule R2.i in Figure 2.1, with n_1 being the node labeled lib and n_2 being the node labeled paper in the left branch of the DAG pattern. Symmetrically, rule R2.ii applies with n_1 being the node labeled figure and n_2 being the node labeled section in the left branch of the DAG pattern.*

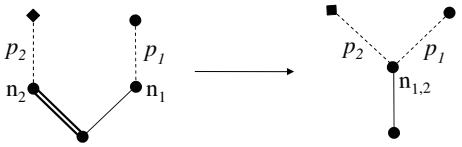
Rule R3.i. This rule triggers if the following conditions hold:

- $p_1 \equiv p_2$,
- each of p_2's nodes has only one incoming main branch edge,
- $\text{TP}_d(p_2)$ contains $\text{TP}_d(p_1)$.

Example 2.3.4. *Notice the application of this rule in Figure 2.1, with n_1 and n_2 being the two nodes labeled paper and the paths p_1 and p_2 consisting of only these nodes.*

Rule R3.ii. This rule triggers if the following conditions hold:

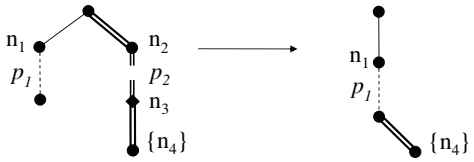
- $p_1 \equiv p_2$,
- each of p_2's nodes has only one outgoing main branch edge,
- $\text{TP}_d(p_2)$ contains $\text{TP}_d(p_1)$.



Rule R4.i This rule triggers if the following conditions hold *for all nodes n_4*:

- n_3 has one incoming main branch edge, all other nodes of p_2 have one incoming and one outgoing main branch edge,
- there exists a mapping from $\text{TP}_d(p_2)$ into $\text{SP}_d(n_1)$, mapping all the nodes of p_2 into nodes of p_1.

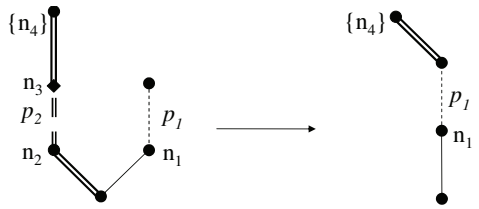
- the path $p_2//n_4$ does not map into p_1.



Example 2.3.5. *The DAG pattern that would be obtained by intersecting some two tree patterns doc("L")/lib/paper/section/.../figure[caption] and doc("L")//lib[.//caption]//section//theorem//... would be subject to R4.i's application, with* p_1 *being the path corresponding to lib/paper/section,* p_2 *being the path corresponding to lib//section, and* n_4 *being the node labeled theorem.*

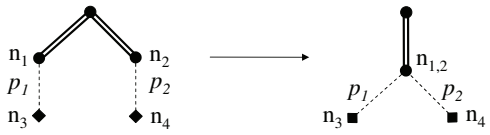
Rule R4.ii. This rule triggers if the following conditions hold *for all nodes* n_4:

- n_3 has only one outgoing main branch edge, all the other nodes of p_2 have one incoming and one outgoing main branch edge,
- there exists a mapping from $\text{TP}_d(p_2)$ into $\text{TP}_d(p_1)$, mapping all the nodes of p_2 into nodes of p_1.
- the path $n_4//p_2$ does not map into p_1.



Rule R5. This rule triggers if the following conditions hold:

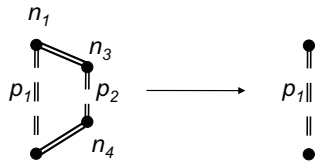
- n_3, n_4 have only one incoming main branch edge, all other nodes of p_1 and p_2 have one incoming and one outgoing main branch edge,
- $\text{TP}_d(p_1)$ and $\text{TP}_d(p_2)$ are similar.



Example 2.3.6. *The DAG pattern that would be obtained by intersecting some two tree patterns doc("L")//lib/paper[.//caption]/section//... and doc("L")//lib[.//figure]/paper/section//... would be subject to R5's application, with the paths* p_1 *and* p_2 *corresponding to the lib/paper/section parts of the queries.*

Rule R6. This rule triggers if the following conditions hold:

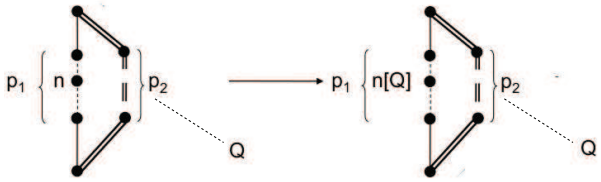
- the nodes of p_2 have only one incoming and one outgoing main branch edge,
- there exists a mapping from $\text{TP}_d(p_2)$ into $\text{SP}_d(n_1)$, such that the nodes of p_2 are mapped into nodes of p_1.



Example 2.3.7. *Notice the application of this rule iFigure 2.1, with p_1 and p_2 corresponding to the two paths* paper//section//figure *in parallel.*

Rule R7. This rule triggers if the following conditions hold:

- the nodes of p_2 have only one incoming and one outgoing main branch edge,
- Q is a /-predicate attached to a node in p_2, such that its presence on the node n would verify the condition of extended skeletons
- for all mappings ψ of p_2 into p_1, for d' being the pattern obtained from d by collapsing each $n' \in p_2$ with $\psi(n')$, *pattern*$(\lambda_d(n)[Q])$ has a root-mapping into $\text{SP}_{d'}(n)$.



Example 2.3.8. *The DAG pattern that would be obtained by intersecting some two tree patterns* doc("L")/lib/section/section/section[figure]/image *and* doc("L")//section[figure]/section[figure]//image *would be subject to R7's application, with predicate Q being* [figure] *and the node n corresponding to the second node labeled* section *in the former view. Note that only after adding Q on node n R6 can apply, removing the branch from the latter view and yielding a tree pattern* doc("L")/lib/section/section[figure]/section[figure]/image.

Figure 2.1 shows how the unfolding of the intersection of views v_1 and v_2 from Example 2.1.15 is rewritten into a prefix of q_2 (Figure 2.1.(c)).

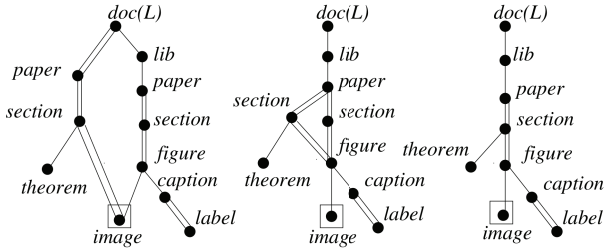


Figure 2.1: Running the rules on Example 2.1.15.

While in next sections we will show conditions for the rule-based algorithm to produce indeed the dominant interleaving, we note hereafter an essential property of APPLY-RULES, of always producing as output a DAG d' equivalent to the input DAG d:

Proposition 2.3.9. *Let d be a DAG and $d' =$ APPLY-RULES(d). Then $d' \equiv d$.*

Proof. For the first six rules, we point the reader to the corresponding proofs in [56]. For rule R7, notice that its conditions imply that in any possible interleaving of d, in particular in any interleaving of the parts p_1 and p_2, the predicate $[Q]$ will be verified at the position where it is copied. This means that each of the interleavings of d' is equivalent to the corresponding interleaving of d, and accordingly that d' and d are equivalent. □

2.3.3 Complexity of APPLY-RULES

We will show hereafter that the complexity of APPLY-RULES is *polynomial in the size of the input DAG*. To this purpose, we first recall a result from [16]:

Lemma 2.3.10. *The rewriting of a DAG d using* APPLY-RULES *always terminates, and it does so in* $\mathrm{O}(|\mathrm{NODES}(d)|^2)$ *steps.*

The proof of the above result is a direct adaptation of the one provided in [56], by further noting that the number of applications of the newly introduced rule R7 is bounded by the combination of nodes and predicates.

We further claim that each rule's testing and application can be achieved in polynomial time. Note that this particular property was not investigated in the previous analysis in [16] and [56]. Our refinement of the rule-based algorithm stems in part from the development of this analysis, leading to the replacement of two of the rules by a new, provably polynomial one, as well as to the novel design of a polynomial testing procedure for rule R6.

Lemma 2.3.11. *The complexity of testing and applying the rules of* APPLY-RULES *is bounded by a polynomial in the size of the input DAG.*

Proof. It is easy to show that rule application is polynomial. We focus in the following on proving polynomial complexity for the testing procedures.

We claim that each of the rules R1-R5 can be tested in polynomial time. For the rules R2-R5, this is based on the *unicity* of the paths involved and on testing the existence or non-existence of

mappings, which can be done in polynomial time. Similarity can also be tested in polynomial time, since the number of patterns p_{12} to be considered is linear in the size of the two /-patterns p_1 and p_2. Note that unicity of /-paths is ensured by the repeated application of rule R1 after other rules' application, part of the our refinement of the rule-based algorithm.

We further exhibit a polynomial testing procedure for R6. Indeed, the test for R6 is more involved, since part of its input, namely the p_1 candidates, is not easily identifiable. To exhibit a polynomial testing strategy, we start from the essential observation that it is sufficient to *test the existence of such candidates* and to handle them *implicitly*, contrary to p_2 candidates which can be found uniquely according to the rule's description.

Given a p_2 candidate, any mapping of p_2 nodes in the DAG rooted at node n_1 will also determine at least one such path p_1; one then only needs to keep ensure that the images of nodes of p_2 in a mapping from p_2 to the subpattern rooted at p_1 are limited to the subgraph induced by n_1 and the common descendant of n_1 and p_2's nodes. We argue that the computation of such a restricted mapping is polynomial, involving an adaptation of the classic dynamic programming procedure. As soon as such restricted mapping exists, we can infer the existence of some p_1 and safely remove p_2, according to the rule.

We discuss next how R7 can be tested in polynomial time. Indeed, the predicate Q can have the following form (Figure 2.2): a /-path $l_1/\ldots/l_k$ followed by either (a) one or more //-edges, (b) one or more //-edges and one or more /-edges, or (c) one or more /-edges. In other words, l_k denotes the highest node having either several outgoing edges (of either kind) or one outgoing edge, of the // kind.

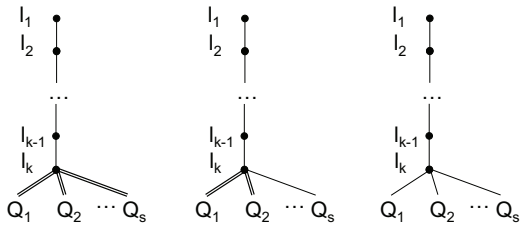


Figure 2.2: The possible configurations for predicate subtree Q.

Case a. If Q is of the first kind, since at node n in p_1 the predicate Q would verify XP_{es}, it means that n is followed by a main branch that is incompatible with $l_1/l_2/\ldots/l_k$. Let $l_1/\ldots/l_{k'}$, for $1 \leq k' < k$, be the maximal prefix that is compatible with the main branch (if one exists). This means that the main branch below n starts by a sequence of labels $l_1/\ldots/l_{k'}/l$, where $l \neq l_{k'+1}$.

For Q to hold at n in each interleaving of p_2 with p_1, it means that in it we have either:

1. Q (or a predicate into which Q can map) attached to n itself (i.e. we do not need the main branch descendants of n and their predicates), or

2. the predicate $l_2/\ldots/l_k[Q_1]\ldots[Q_s]$ (or a predicate into which it can map) attached to n's main branch child n' (i.e. we do not need the main branch descendants of n' and their predicates), or

3. the predicate $l_3/\ldots/l_k[Q_1]\ldots[Q_s]$ (or a predicate into which it can map) attached to n's main branch descendant at distance 2, n'' (i.e. we do not need the main branch descendants of n'' and their predicates), or so on, ...

(k') the predicate $l_{k'+1}/\ldots/l_k[Q_1]\ldots[Q_s]$ (or a predicate into which it can map) attached to n's main branch descendant at distance k', $n^{(k')}$, (i.e. we do not need the main branch descendants of $n^{(k')}$ and their predicates).

Accordingly, in order to test that Q holds at n in each interleaving of p_2 with p_1, we need to test the *non-existence* of a mapping from p_2 into p_1 that would not bring a predicate as the ones described above on any of the nodes $n, n', n'', \ldots, n^{(k')}$. This test can be done in polynomial time, top-down and one token at a time, by choosing as long as possible for each token of p_2 the highest-possible image that does not contribute any predicates like the ones described above.

Case b. This case is similar to the previous since we have the same setting, i.e., n is followed by a main branch that is incompatible with $l_1/l_2/\ldots/l_k$ and we have at most a prefix of it $l_1/\ldots/l'_k$, for $1 \leq k' < k$, that is compatible (if such a prefix exists).

Case c. If n is followed by a main branch that is incompatible with $l_1/l_2/\ldots/l_k$, then the same reasoning of the two previous cases applies here as well. Otherwise, for Q to hold at n in each interleaving of p_2 with p_1, it means that in each interleaving we have either:

1. Q (or a predicate into which Q can map) attached to n itself (i.e. we do not need the main branch descendants of n and their predicates), or

2. predicate $l_2/\ldots/l_k[Q_1]\ldots[Q_s]$ (or one into which it can map) attached to n's main branch child n' (i.e., we do not need the main branch descendants of n' and their predicates), or so on, ...

(k) the predicate $l_k[Q_1]\ldots[Q_s]$ (or a predicate into which it can map) being present (as a predicate) on n's main branch descendant at distance k, $n^{(k)}$, (i.e. we do not need the main branch descendants of $n^{(k)}$ and their predicates), or

(k+1) all the predicates $[Q_1], \ldots, [Q_s]$ verified at n's main branch descendant at distance $k+1$, $n^{(k+1)}$.

So a similar test for the non-existence of a mapping has to be done, but with some minor adjustments. Top-down, we will chose a mapping image for each token of p_2 into p_1, as long as we do not arrive at the position of $n^{(k+1)}$ or below it (i.e. we will chose an image for a token if it does not overpass this position and does not contribute predicates like the ones described by the items (1) to (k) above). Then, for the remaining suffix of p_2, we check the existence of a mapping for it that would (i) not contribute predicates like the ones given in conditions (1) to (k), and (ii) would not contribute *all* the predicates of the last condition, i.e., that there is a mapping for the remaining part of p_2 in the remaining part of p_1 s.t. among $Q_1, \ldots, Q_s$ there is at least one predicate Q_i which will not be verified at $n^{(k+1)}$ after coalescing p_2's nodes with their mapping images. This can be seen as a recursive call, that can be run for each Q_i individually, and will take us back to the three cases depending on the shape of Q_i. (Note that all the predicates $Q_1, \ldots, Q_s$ at node $n^{(k+1)}$ on p_1 will verify the condition for extended skeletons.)

A dynamic programming approach can be used to perform all these tests in polynomial time, based on the to-be-mapped suffix of p_2, the target suffix of p_1 and the predicate to be tested (it is not necessary to perform the test several times for a given such triple). □

2.3.4 Using APPLY-RULES for union-freeness, equivalence and rewritings

Since APPLY-RULES preserves equivalence, it can be directly used as a brick for solving the union-freeness problem for a DAG d as well as the equivalence problem as follows:

UF(d)

```
1   p1 ← APPLY-RULES(d);
2   p2 ← DOMINANT_INTERLEAVING(p1);
3   if p2
4     then return p2
5     else  return ∅
```

EQUIV(d, p)

```
1   p1 ← APPLY-RULES(d);
2   p2 ← DOMINANT_INTERLEAVING(p1);
3   if p2 and p2 ≡ p
4     then return TRUE
5     else  return FALSE
```

Also, REWRITE can be changed to incorporate APPLY-RULES. We show below the resulting modified version of REWRITE , detailing and clarifying its previous presentation in [16]:

REWRITE($q, \mathcal{V}$)

```
1   for p a lossless prefix of pattern(q)
2       do
3           r ← BUILDINITREWITECANDIDATE(p, V)
4           d ← dag(unfold(r))
5           p1 ← APPLY-RULES(d);
6           p2 ← DOMINANT_INTERLEAVING(p1);
7           if p2 and p2 ≡ p
8             then return compensate(r, q, OUT(p))
9   return fail
```

While the soundness and completeness of the above algorithms is straightforward, the problems they solve stay hard, and DOMINANT_INTERLEAVING can be very expensive. One can however use APPLY-RULES to design *sound, polynomial versions of the above procedures*. We give below the resulting algorithms for union-freeness and equivalence:

EFFICIENT-UF(d)

1 $p_1 \leftarrow$ APPLY-RULES(d);
2 if p_1 is a tree
3 **then return** p_1
4 **else return** ϕ

EFFICIENT-EQUIV(d, p)

1 $p_1 \leftarrow$ APPLY-RULES(d);
2 **if** p_1 is a tree and $p_1 \equiv p$
3 **then return** $TRUE$
4 **else return** $FALSE$

We further recall the sound algorithm described in [16] for solving the rewriting problem:

EFFICIENT-RW$(q, \mathcal{V})$

1 **for** p a lossless prefix of *pattern*(q)
2 **do**
3 $r \leftarrow$ BUILDINITREWITECANDIDATE$(p, \mathcal{V})$
4 $d \leftarrow$ *dag*(*unfold*(r))
5 $p_1 \leftarrow$ APPLY-RULES(d);
6 **if** p_1 is a tree and $p_1 \equiv p$
7 **then return** compensate$(r, q,$ OUT$(p))$
8 **return fail**

2.4 Achieving PTIME completeness

In the light of the hardness results of Theorem 2.1.16 (and Theorems 2.2.2 and 2.2.4), one cannot hope of obtaining sound and complete polynomial algorithms for the problems we analyse.

The approach taken by [16] is that of identifying *restrictions* for which completeness is efficiently achievable in solving the rewriting problem. We dedicate this section to recalling and enriching these completeness conditions, showing also how they apply to the equivalence and union-freeness problems.

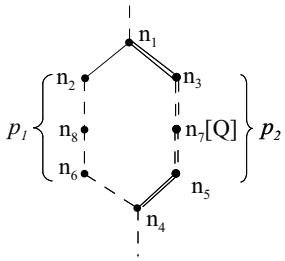
2.4.1 Completeness in PTIME for $XP^{\cap-simple}$ (XP_{es}) DAGs

This subsection will focus on DAGs obtained by an $XP^{\cap-simple}$ intersection of extended skeleton queries, that is, queries in the fragment XP_{es}. We will present below a refinement of the results claimed in [16] and proven in [56], showing how completeness is achievable for such DAGs. We start by showing the following:

Theorem 2.4.1. *Let d be a DAG in $XP^{\cap-simple}$ (XP_{es}), such that one of the patterns intersected is a /-pattern. If d is union-free then* APPLY-RULES *rewrites it into a tree.*

Proof. Let $v_2, v_3, \ldots v_n$ be the n XP_{es} patterns intersected in d, where v_1 has only one token. Without loss of generality, we can consider that $v_2, v_3, \ldots v_n$ have more than one token. This is because intersecting several /-patterns reduces easily to a single pattern by repeated applications of rule R1.

Supposing that d', the output of the rewriting algorithm, it not a tree, we can then claim the existence in d of a subpattern sd as in the figure below, such that the /-path p_1 is the unique /-path between n_1 and n_4 (part of v_1) and the intermediate nodes on p_2 (part of one of the other patterns) have only one incoming and one outgoing main branch edge.



Indeed, note that the existence of this configuration is ensured by the application of the rules in the rule set corresponding to the presence of a /-pattern R1, R2, R3, R4, R6, R7. In particular, if at least one of $v_2, \ldots, v_n$ has a root token with more than one main branch node, this token will be collapsed by R1 with the corresponding part of v_1; any subsequent collapsing of this main branch with the main branches of the other patterns will, according to the definition of R2, R3 and R4, not go beyond this token. If none of the multi-token patterns posesses such root token, then the initial configuration is preserved, modulo some applications of R6 that remove entire branches. The same reasoning stands for result tokens.

Since R1 does not apply, we further infer that the first and last edges on p_2 are //-edges. Let us assume that d is union-free. Since R6 doesn't apply on sd, it means that the tree corresponding to p_2 does not fully map into the subpattern corresponding to p_1. This means that a dominant interleaving i, built by some choice ψ of mapping p_2's main branch nodes into p_1's main branch nodes (note that this is is the only possible format of the dominant interleaving because of p_1 being a /-path), must for at least some node n_7 in p_2 and /-predicate Q attached to it (note that for extended skeletons we cannot have //-predicates attached to the main branch nodes) collapse n_7 with $n_8 = \psi(n_7)$ of p_1, even though *pattern*$(\lambda_d(n_7)[Q])$ does not map into $\text{SP}_{d'}(n_8)$. In this interleaving, Q will then further be present at n_8.

Since Q respects the extended skeletons condition on n_7, it means further that it respects this condition when added to n_8 (because the /-paths following the nodes must be indetical) . But then rule R7 is supposed to have added Q on n_8, if it held in all interleavings. According to the above this is not the case (i.e. or else rule R6 would apply), which then means that there exists at least one interleaving i' of d such that Q does not hold at n_8 in i'. Then i cannot map into i', and thus i' cannot be contained in i. It follows that therefore i cannot be a dominant interleaving, and accordingly d cannot be union-free.

Note that the absence of rule R5 from the set of rules applicable in such setting (as presented in our refinement of APPLY-RULES) is important for the above proof to hold. Indeed, an appli-

cation of rule R5 on two of the multi-token views could prevent the existence of a subpattern as the one exhibited above, by adding edges to the nodes of p_2 and thus preventing the conditions of applicability of the rules R6 and R7. □

We then recall a result from [16] and [56]:

Theorem 2.4.2. *Let d be a DAG in $XP^{\cap-simple}$ (XP_{es}), such that all the intersected patterns are multi-token. If d is union-free then* APPLY-RULES *rewrites it into a tree.*

The proof of the above result is provided in [56]. Note that this proof uses only the corresponding set of rules for this case presented in our refinement of APPLY-RULES.

Putting together the above results and the equivalence preserving property of APPLY-RULES, we can prove the following claim, as adapted and clarified from [16]:

Theorem 2.4.3. *Let d be a DAG in $XP^{\cap-simple}$ (XP_{es}). Then d is union-free iff* APPLY-RULES *rewrites it into a tree.*

Accordingly, we can characterize the union-freeness and equivalence problems as follows:

Corollary 2.4.4. *For a DAG d in $XP^{\cap-simple}$ (XP_{es}), the union-freeness problem is PTIME and the algorithm* EFFICIENT-UF *is complete.*

Corollary 2.4.5. *Deciding equivalence between a DAG pattern d in $XP^{\cap-simple}$ (XP_{es}) and an XP query is PTIME and the algorithm* EFFICIENT-EQUIV *is complete.*

We cannot however directly extend the result above to the rewriting problem. The reason is that the compensation applied on the views by BUILDINITREWRITECANDIDATE may violate the extended skeletons condition on the resulting compensated patterns, even if the input query and views are in XP_{es}. [56] proposes an adjustment in order to account for such cases, that is further adaptable for our relaxation of XP_{es}. With this adjustment, [56] then proves the claim from [16] that the rewriting problem is PTIME for queries and views in XP_{es}. We provide below a stronger result, by showing that as soon as the input query is in XP_{es}, the rewriting problem is PTIME for *views in XP*, thus extending the previous result targeting only views from XP_{es}.

2.4.2 Completeness in PTIME for XP_{es} queries

We hereafter show how Theorem 2.4.3 can be used to derive additional PTIME results for two of the problems investigated here, namely the DAG-tree equivalence problem and the rewriting problem. As a side result of the analysis hereafter, we enhance the completeness conditions in [16] and [56].

As in [16] and [56], the extended skeleton of a pattern p is denoted by $s(p)$ and represents the pattern obtained by pruning out all the //-subpredicates violating the XP_{es} condition. We start by recalling a result from [56]:

Lemma 2.4.6. *Let d be a DAG pattern in $XP^{\cap-simple}$. If $s(d)$ is not union-free then d is not union-free.*

We further emphasize the strong link that exists between a tree pattern in *XP* and its extended skeleton, regarding *containment* in a tree pattern in XP_{es}:

Proposition 2.4.7. *Let p be a tree pattern in XP and q a tree pattern in XP_{es}.*
Then $p \sqsubseteq q$ iff $s(p) \sqsubseteq q$

Proof. Since $p \sqsubseteq s(p)$, if $s(p) \sqsubseteq q$ then obviously $p \sqsubseteq q$. The rest of the proof is a direct adaptation of part of the proof of Lemma 2.4.6 in [56].

Indeed, if $p \sqsubseteq q$, suppose that $s(p) \not\sqsubseteq q$. Recall that for tree patterns containment is equivalent to containment mapping in the opposite direction. Any containment mapping from q into p should then use at least one of the //-subpredicates of p, st', which is not in $s(p)$. But for the //-subpredicate st in q that maps in st', its incoming /-path as well as the /-path following the corresponding main branch node must be identical to their image in p. It follows that if st' violates the extended skeleton condition, then st violates the extended skeleton condition, thus leading to a contradiction. Therefore $s(p) \sqsubseteq q$. □

We further note [56] the strong correspondence between the interleavings of a DAG d and those of its extended skeleton $s(d)$:

Proposition 2.4.8. *Let d be a DAG pattern in $XP^{\cap-simple}$.*
Then for each $p_i \in interleave(d)$ there exists $p'_i \in interleave(s(d))$ such that $s(p_i) = p'_i$ and reversely, for every interleaving $p'_i \in interleave(s(d))$ there exists $p_i \in interleave(d)$ such that $p'_i = s(p_i)$.

Based on the above and Lemma 2.1.12, we can directly extend Proposition 2.4.7 to DAGs:

Lemma 2.4.9. *A DAG pattern d in $XP^{\cap-simple}$ is contained in a tree pattern q in XP_{es} iff $s(p) \sqsubseteq q$.*

Let us examine the above results. They do not provide an additional complete characterization of the union-freeness problem for DAGs in $XP^{\cap-simple}$ (that is, intersecting queries in *XP*). However, they *do allow characterizing the equivalence problem for such DAGs and a tree in XP_{es}*, by essentially *reducing this problem to the union-freeness problem of s(d)*. We can thus modify EFFICIENT-EQUIV as follows:

EFFICIENT-EQUIV(d, p)

1 $p_1 \leftarrow$ APPLY-RULES(s(d));
2 **if** p_1 is a tree and $p_1 \sqsubseteq p$ and $p \sqsubseteq d$
3 **then return** $TRUE$
4 **else return** $FALSE$

We claim that the above algorithm is polynomial. Indeed, containment of a tree pattern in a DAG pattern is witnessed by containment mapping, which can be tested in PTIME. Furthermore, by Lemmas 2.4.9 and 2.4.6 and Theorem 2.4.3, we can infer that the above algorithm is sound and complete and state the following:

Theorem 2.4.10. *Equivalence between an $XP^{\cap-simple}$ DAG pattern d and an XP_{es} tree pattern p is PTIME and the algorithm* EFFICIENT-EQUIV *(with the changes above) is complete.*

Proof. Indeed, by Lemma 2.4.6, if $s(d)$ is not union-free then d cannot be union-free, so the equivalence wouldn't hold. But according to Theorem 2.4.3, $s(d)$ is union-free iff p_1 is a tree. So if p_1 is not a tree then the equivalence cannot hold. On the other hand, if p_1 is a tree, since p_1 is equivalent to $s(d)$, by Lemma 2.4.9, $d \sqsubseteq p$ iff $p_1 \sqsubseteq p$. □

We extend the above to the rewriting problem and produce the following modified version of EFFICIENT-RW , which is, by the same criteria, sound and complete:

EFFICIENT-RW$(q, \mathcal{V})$

1 **for** p a lossless prefix of *pattern*(q)
2 **do**
3 $r \leftarrow$ BUILDINITREWITECANDIDATE$(p, \mathcal{V})$
4 $d \leftarrow dag($*unfold*$(r))$
5 $p_1 \leftarrow$ APPLY-RULES(s(d));
6 **if** p_1 is a tree and $p_1 \sqsubseteq p$
7 **then return** $\mathrm{compensate}(r, q, \mathrm{OUT}(p))$
8 **return fail**

Note that, as emphasized earlier, we do not need the containment test $p \sqsubseteq d$ (although it is polynomial), since by construction of the rewrite plans, the containment holds. We can then state the following:

Theorem 2.4.11. *The rewriting problem for queries in XP_{es} and views in XP is PTIME and the algorithm* EFFICIENT-RW *(with the changes above) is complete.*

2.4.3 Completeness in PTIME for $XP_{//}$ akin patterns

To further extend the completeness results beyond XP_{es}, [16] considers the fragment $XP_{//}$. While it is shown that the rewriting problem for queries and view in $XP_{//}$ is coNP-complete, [16] further considers an additional restriction, regarding *$XP_{//}$ akin patterns*.

Two (or several) tree patterns are said to be akin ([16]) if their root tokens have the same main branch codes. We recall below the main result claimed in [16] and proven in [56] regarding $XP_{//}$ akin patterns:

Theorem 2.4.12. *For a DAG pattern in $XP^{\cap-simple}$ intersecting $XP_{//}$ akin patterns, if $\mathtt{dag}(d)$ is union-free then* APPLY-RULES *rewrites it into a tree.*

Note that the prove of the above in [56] holds for our refinement of the rule-based algorithm. Following this result, we can further infer the following results for the union-freeness and equivalence problems:

Corollary 2.4.13. *For a DAG pattern in $XP^{\cap-simple}$ intersecting $XP_{//}$ akin patterns, the union-freeness problem is PTIME and the algorithm* EFFICIENT-UF *is complete.*

Corollary 2.4.14. *For a DAG pattern in $XP^{\cap-simple}$ intersecting $XP_{//}$akin patterns, and a query p in XP, the DAG-tree equivalence problem is PTIME and the algorithm* EFFICIENT-EQUIV *is complete.*

We also recall the result from [16] regarding the rewriting problem:

Corollary 2.4.15. *For queries and views in $XP_{//}$,* EFFICIENT-RW *is complete, provided there is at least one rewriting r such that the patterns intersected in unfold(r) are akin.*

2.5 Implementation and optimizations

The main motivation behind the work in this chapter was the desire to achieve practical performance for the presented rewriting techniques. The review of previous work under such pragmatically-driven considerations is indeed the source of the presented refinements in the rule-based algorithm, towards ensuring polynomial complexity and enhancing completeness. The first tentative implementation of the rewrite algorithms (the sound and complete algorithm, as well as the sound polynomial one) revealed furthermore a vast range of possible optimizations toward ensuring practical performance.

A first category of such improvements concerns the construction of candidate plans. Indeed, note that due to the compensation steps, many redundant predicates may be present on nodes of the initial plan. These predicates can be pruned out before the APPLY-RULES subroutine, in this way making the various mapping tests for rule applications lighter. Note also that in the case of $XP_{//}$ queries and plans involving akin $XP_{//}$ views, completeness is guaranteed for EFFICIENT-RW ; on the other hand, the strategy of constructing rewrite candidates can cause an important redundancy in the candidate plans. We have therefore identified as beneficial the testing of sub-plans involving akin patterns for equivalence to the prefix considered (if this is the case, then global plan equivalence follows). To generally minimize the redundancy in the plans, we can also employ containment tests that remove compensated views that participate in the candidate plans and whose results contain the results of some other compensated views. Finally, note that we can use some efficient tests to detect plans that cannot be equivalent to the input query and can be discarded before the APPLY-RULES subroutine. For example, in the case of an input queries with only /-edges in the main branch, a view having that same main branch must be available. Similar tests on the main branches of the root and output tokens of the query and the plan's views can be used to discard plans.

While the optimizations in the construction of candidate plans are useful in gaining performance, the most complex and challenging task in order to achieve practically relevant run speed consisted in the implementation of the rule-based algorithm. Each rule testing and application can be indeed seen as a "sub-algorithm" on its own. A central brick of the testing procedures for the rules consists in the mapping computations between tree patterns, for which we have implemented and optimized the dynamic programming approach in [53]. We have further developed optimized implementations for a range of mapping variations, such as those from DAG patterns to tree patterns, where the DAG nodes are stored in topological order, as well as mappings from paths to DAGs, such as those needed for the rules R4 and R6. The polynomial testing strategies for the rules R6 and R7, which we present with our complexity analysis, were developed

in tight connection with the performance requirements of our implementation. To importantly improve the runtime speed for rule R2, we changed the test from collapsible nodes from its formal definition, based on the observation that we can in fact bypass the computation of tentative DAGs and simply compare the incoming and outgoing /-paths for the tested nodes. If these paths are the same up to a common ancestor or descendant or until the shortest of the two sizes if such "meeting point" doesn't exist, we can conclude to applicability of R2, without applying the formal immediate unsatisfiability condition. For the rule R4, we have further mutualized the computation of mappings from parts of p_2 to parts of p_1, so as to detect candidate paths for the application of this rule as early as possible.

An important number of general adjustments that proved to be useful for the overall performance are related to the usage of dedicated data structures, such as adjacency lists for incoming and outgoing main branch edges, predicates, child and descendant edges, as many of the rules involve iterating on specific children types, as well as lists of topologically-sorted nodes, built with the candidate plan and updated only when needed, after certain rules were applied. The pre-computation/update-only-when-needed is a general optimization direction that we further used for structures such as mapping matrixes and paths.

2.6 Experiments

We performed our experiments on an Intel(R) Core(TM) i7-2760QM@2.40GHz machine, with 8G of RAM and the Ubuntu 11.10 operating system. We evaluated the performance and scalability of our the procedures, focusing on:

- the *rewrite time*, i.e., the time necessary to find an equivalent rewriting, when one exists
- the improvements on *evaluation time*, i.e., the comparison between the evaluation time of the input query over the data, on one hand, and the rewrite time cumulated with the evaluation time of the rewriting, over the view documents, on the other hand.

In the space of analysis, we looked at how these two performance indicators vary with the size and the type of input queries (w.r.t. several *XP* fragments discussed in the paper), the size of the view set that may give a rewriting, and the size of the input document.

2.6.1 Documents, queries and views

Our experimental setup was guided by our focus on measuring rewrite time as well as improvements on evaluation time, as well as the intention to stress-test our implementation for performance evaluation purposes. We thus needed:

- queries (spanning the *XP* fragments analyzed in our theoretical study) and views to materialize
- a set of documents the queries and views would apply to,
- *the ability to scale* query, view set and document sizes, for performance assessment.

Given our needs in terms of variation of query types, we could not benefit from existing benchmarks or real-life settings publishing queries and views. Therefore we designed *our own synthetic query and views generator*, suiting our testing purposes. Starting from a given XML input document, this generator produces queries and views over that document (i.e. yielding a non-empty result), controlling their structure, number and size, as well as pair-wise containment.

While our synthetic queries and views generator can be plugged on any XML document, our need to scale with the document size limited the usefulness of existing XML documents. We have therefore adopted in our experiments the extensively cited XMark document generator [63]. This generator allows varying the size while ensuring similar structure and properties across the XML documents it produces. Three input documents were generated with the XMark generator, of sizes 41KB, 91MB and 18GB. On each of the documents, we used our custom generator to produce input queries and view sets.

We generated, for each of the documents, input queries of main branch size 5, 7 and 9 (the XMark documents have a maximal depth of 11). We considered input queries from three categories: XP_{es}, beyond XP_{es} but in $XP_{//}$, and in XP but beyond $XP_{//}$, i.e., without restrictions. Each possible pairing of main-branch size and category was used to generate 10 random input queries, for a total of 90 input queries. Importantly, these queries were generated from the data, in a way that ensures that they all have a *non-empty result* on the three input documents. This was to avoid meaningless evaluation time measurements and to preclude the case when an alternative detection of unsatisfiability would shortcut the rewrite time.

For the generation of views, to each of the input queries we associated five randomly generated view sets of variate size, namely consisting of 40, 80, 160, 320, or 640 views, for a total of 450 view sets. We had the following guidelines in the generation of view sets:

1. While we wanted many views, we wanted to control the percentage of views that would be useful in a rewriting; more precisely, all the view sets consisted of 10% useful views (for the rewriting), while the remaining 90% were useless (i.e., they did not map in the input query)[2].

2. We wanted views that were not equivalent to the input query nor a prefix thereof, and did not allow single-view rewritings, in order to test precisely the targeted multiple-view rewriting problem.

Note that, although all views have a non-empty result by construction, the size of their result and their selectivity could vary significantly and were not controlled in the generator. Other aspects that were not controlled by our query generator were (i) the overall size of input queries (only the size of the main branch was chosen), (ii) the overall size of the views, and (iii) the overall size of the candidate plans.

2.6.2 REWRITE vs. EFFICIENT-RW

As a first experiment, through the random generation of sets of views, we took a first step towards understanding how often one may lose completeness in practice by employing EFFICIENT-RW

[2]We adopted this $10\% - 90\%$ ratio as a reasonable one for most practical scenarios.

rather than REWRITE . This is important for input queries from $XP_{//}$ and *XP*, as the computation of interleavings – potentially exponentially many – is expected to represent the main overhead in the search for a rewriting.

To this end, the random generation flow was the following: (a) a set of views would be generated, for a given input query (in $XP_{//}$ or *XP*) and a given view set size, and (b) REWRITE would be run (in its version employing APPLY-RULES) *within a limit of 30 minutes of execution time*.

This experiment gave us valuable insight : within the time limit, for the 300 configurations tested, we obtained no view set that did provide an equivalent rewriting, *but only by performing interleavings, after* APPLY-RULES. Note that this computation of interleavings is the essential difference between REWRITE and EFFICIENT-RW . In about a third (112) of the tested cases, the interleaving computation process reached the timeout without concluding.

Our experiment shows on one hand that completeness of EFFICIENT-RW extends in practice beyond the considered restrictions. Moreoever, according to our evaluation, the significant amount of time spent in interleaving computation does not show essential utility in practice. Indeed, for the timeout cases where one cannot not decide on the completeness of EFFICIENT-RW the very high running time importantly lowers the interest of such computation in practically relevant scenarios.

We have consequently continued our empirical evaluation using the EFFICIENT-RW rewriting procedure. We further focused on sets of views on which EFFICIENT-RW is guaranteed to provide a rewriting for the input query, and on measuring how fast such rewriting is found and how beneficial the rewriting proves in practice.

2.6.3 Rewrite time

For this set of measurements, for each input query size and category, for the corresponding 10 input queries, we recorded the average time to find a rewriting using EFFICIENT-RW for each possible size of the view set (among the 5 sizes given previously). We present our measurements for the rewrite time in Figure 2.3. We give one set of results (a sub-figure) for each query length. In each sub-figure, we give five groups of three columns. A group corresponds to one possible size of the view set, and in each group the first column corresponds to XP_{es} input queries, the second column corresponds to $XP_{//}$ input queries, and the third column corresponds to input queries without restrictions.

We can draw several important conclusions from the results of in Figure 2.3. First, our implementation of EFFICIENT-RW can process efficiently, in a fraction of a second, queries of significant size – up to 9 nodes in the main branch, with $3-4$ predicates in average on each main branch node and with predicates of average depth of 3 – and view sets of significant size as well (order of hundreds). Note that the measurements follow closely a linear progression with respect to the size of the view set. With respect to varying the query size, the observed progression is even less pronounced – for example, for queries without restrictions, from 110ms to 210ms to 250ms.

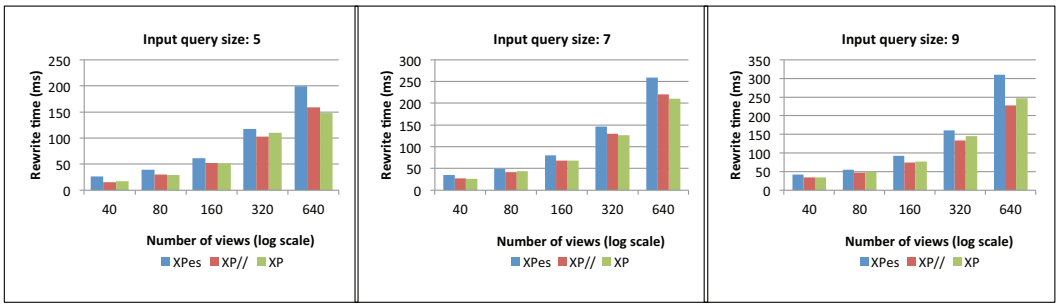


Figure 2.3: Rewrite time results.

2.6.4 Evaluation time

Regarding evaluation time, we compared the time necessary to evaluate an input query over the input documents with the time necessary to build, test and then evaluate the rewriting over the view documents. For each document size, we have generated one maximum-size query in *XP* (9 nodes in the main branch), and further generated for its original, XP_{es} and $XP_{//}$ versions, sets of views of increasing size such that, as in the previous experiment, a rewriting is available and produced by EFFICIENT-RW with the considered views. Query evaluation was done using the SAXON query engine (http://saxon.sourceforge.net), which we extended with the Id-based JOIN functionality across multiple documents (the view documents), as SAXON's ability to perform this task was incomplete.

We present our measurements in Figure B.7. We give one set of results (a sub-figure) for each document size. As before, in each sub-figure, we give five groups of three columns, with one group for each possible size of the view set. Since the time necessary to run the input query over the input documents does not depend on the views and further stays roughly the same for its three versions, it is represented by a horizontal line in the plot.

A first important aspect to be noted in Figure B.7 is that, over all input documents, the time necessary to evaluate the rewriting is smaller than the one for the input query, for all sizes of view sets. Moreover, the evaluation time based on view documents exhibits a linear progression and, overall, remains quite low.

One can note the intuitive trend indicating that the larger the set of views in the rewriting, the less important the performance benefit over the original query plan (note that we measured the plans consisting of all the useful views). In our results, this trend is enforced by the way we set up the experiments, doubling at each step the number of views that were applicable in a rewriting (while this seems to be an unlikely scenario in practice, it represents a suited stress test). As a partial explanation of such trend, note also that in our experimental configuration many views means inevitably many opened documents, hence the overhead related to managing them, which for SAXON starts being noticeable. Note also that within one group of columns, the differences in evaluation time between the three categories of queries are mainly due to the variations in terms of selectivity and view documents' size. For instance, on the smallest

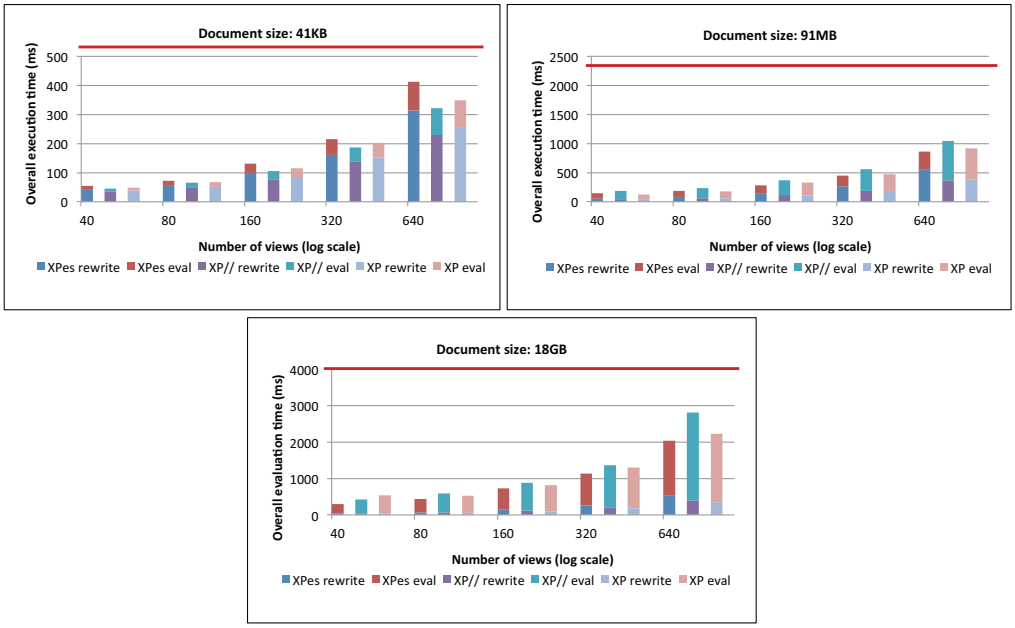


Figure 2.4: Global time results (rewrite time plus evaluation time).

document, the views randomly generated for the XP_{es} version of the query were significantly less selective, yielding view documents almost two times larger than the ones corresponding to the other two categories. Similar differences could be observed for the second document, between views for the XP_{es} and $XP_{//}$ queries on one hand, with larger view documents, and the views for *XP* queries on the other hand.

2.6.5 Discussion

Our main conclusions from this experimental evaluation are the following:

- Our optimized implementation of the refined and polynomial algorithm EFFICIENT-RW scales to large sets of views, with rewrite time under one second in all cases. Moreover, the rewrite time represents a small percentage of the evaluation time. At the same time, there are many scenarios (e.g., with security views) where rewriting is not done for performance purposes, and in which the comparison between rewrite time and evaluation time is immaterial.

- The evaluation of the rewriting, including the rewrite time, is significantly more efficient than the evaluation of the input query; despite the fact that views were generated without controlling their selectivity or how they may "cover" the input query, the rewriting was evaluated two to three times faster than the input query. Note that for stress-test purposes the percentage of useful views was increased exponentially, which is certainly not what one would expect in practice. Finally, the evaluation time depends undoubtedly on the

particular query engine that is used, and it is not clear whether the one that we relied on had an optimal behaviour when handling many opened, large documents.

- The practical benefit of the refined and optimised polynomial rewriting technique is significant. Indeed, while according to theoretical results, for input queries outside XP_{es}interleaving computation is necessary to achieving completeness, our experiments show that completeness extends beyond these restrictions, and furthermore that the benefit of computing interleavings is clearly limited in practice; EFFICIENT-RW then stands out as a good candidate for practical, performance-oriented rewriting scenarios.

2.7 Related Work

Several studies have analyzed the problem of XPath rewriting using only one view [68, 48, 65, 69, 67], possibly in the presence of DTD constraints [6]. Rewriting more expressive queries using views, but without considering intersection, was studied in [18, 30, 58]. Fan et al [34] define views using DTDs instead of queries and study the problem of rewriting an XPath using one view DTD. [2] describes a sound but incomplete algorithm for finding equivalent rewritings as unions of single-view rewritings, for an XPath fragment including wildcards.

[49] describes complete rewriting procedures for multiple views for tree pattern queries with value joins and multiple arity, language where equivalence is intractable and no complete rewriting algorithm can go below the exponential bound. Closure under intersection is analyzed in [10] for various XPath fragments, all of which use wildcard. Satisfiability of XPath is analyzed and proven NP-complete in [41] in the presence of the intersect operator and of wildcards, and also analyzed in [9] for fragments including negation and disjunction, which could together simulate intersection, but lead to coPSPACE-hardness for checking containment. Richer sublanguages of XPath 2.0, including path intersection and equality, are considered in [66], where complexity of checking containment goes up to EXPTIME or higher. A different approach, taken by [39], is to replace intersection by using a rich set of language features, and then try to simplify the expression using heuristics.

We revisit in this chapter the work presented in [16] and detailed in [56], which analyses a fragment of XPath without wildcards, and provides the first complexity analysis for the single-level intersection rewriting problem in this setting. We enrich the previous analysis by showing, structuring and clarifying the links between the rewriting problem and that of deciding the equivalence between a DAG and a tree pattern, as well as the union-freeness problem for a DAG. An essential contribution of [16] and [56] is in investigating efficient rewriting techniques, thus going beyond the stated hardness results. We refine these techniques to ensure their polynomial complexity and improve their completeness, and further optimize them to achieve practical performance, thus showing their potential and utility in practically oriented scenarios.

Conclusions and future directions

We have described in the first and main chapter of this thesis a global and efficient approach for computing all the minimal conjunctive query reformulations of a relational conjunctive query under integrity constraints, by the Provenance-Aware Chase and Backchase algorithm. We have also presented a set of theoretical results guaranteeing the soundness and completeness of $Prov_{C\&B}$. In particular, for the purpose of our reformulation algorithm, we have introduced a novel chase flavour, called the Provenance-Aware Chase, and its underlying Conservative Chase procedure. We have shown how the Provenance-Aware Chase allows us to directly "read-off" minimal reformulations from the result of a chase sequence, and further shown how the Provenance-Aware Chase is particularly suited for cost-based pruning interleaved with the chase steps, thus providing additional speed-up opportunities.

We will continue investigating additional theoretical results regarding the Provenance-Aware Chase (and the underlying Conservative Chase) complexity and termination. Indeed, as sketched with the initial presentation of $Prov_{C\&B}$, additional termination conditions for the Conservative Chase (and accordingly for the Provenance-Aware Chase), beyond weak acyclicity, would allow extending the range of the inputs guaranteeing soundness and completeness for $Prov_{C\&B}$. On the other hand, a refined complexity analysis, both regarding time and space requirements, would allow us to better characterize the practical performance in relation with theoretical properties.

We also intend to investigate the usage of the cost-based pruned version of our algorithm in a unified approach of the chase and backchase phases, thus extending the provenance-aware approach to settings where the chase does not terminate. Furthermore, while the particular provenance flavour that we employ corresponds to minimal-why provenance [14], we aim at analysing the possibility of adapting the Provenance-Aware Chase to other existing provenance flavours, thus enlarging and generalizing its theoretical framework.

On the other hand, recall that we were driven in our design of the $Prov_{C\&B}$ by the pragmatic need of achieving performance. Our experimental results confirm the practical interest of our solution, both concerning its standalone performance and the global benefits attainable by its usage with a DBMS. However, a lot of optimizations are still directly available. As emphasized when presenting our implementation, we exploit therein the analogy between chase steps and query evaluation, and thus can further optimize our approach by refining techniques for constructing the query plans. We estimate that practical performance can also be improved by employing more efficient storage and search structures for boolean formulae, extensively used in the $Prov_{C\&B}$. Furthermore, one can also envision a compact, "folded" version of the provenance formulae. These could be then unfolded in a Datalog-like fashion and solely when "reading-off" minimal reformulations, thus reducing the cost of formulae manipulation and speeding up the

global Provenance-Aware Chase execution. A refined analysis and handling of boolean formulae could also be envisioned not only as a practical performance enhancer but, from a theoretical perspective, as a means to formally reduce the worst-case exponential space bound induced by DNF provenance formulae storage.

We intend to implement these optimizations and investigate, from both a theoretical and practical standpoint, the gain in performance that they can provide. We also intend to improve and widen the applicability of our algorithm towards some of the numerous practical settings requiring view-based rewritings under constraints, such as those mentioned in the introductory section. While our experiments evaluate $Prov_{C\&B}$ in a traditional query processing setting, where partial rewritings are of interest, we estimate that scenarios requiring completeness where the search space for rewritings is very large but few rewritings exist (such as total view based rewritings for access control enforcement or data pricing) provide an ideal applicative setting for $Prov_{C\&B}$.

In the second chapter of this thesis, we revisited the previous work of Cautis, Deutsch and Onose, presented in [16] and detailed in [56], on the problem of rewriting XPath queries using a single-level of intersection of multiple views, enriching the analysis of this problem by showing its connections to the equivalence between DAG and tree patterns, and the union-freeness of a DAG pattern. The main motivation of the work presented in the second chapter was that of investigating and achieving practical performance for the rewriting techniques. To this purpose, we have refined a previously proposed rule-based procedure towards ensuring its polynomial complexity and improving the completeness of the resulting rewriting algorithm. We have further provided a range of optimizations necessary for achieving practical performance, and included these refinements and optimizations in a complete implementation, which we have then evaluated experimentally.

While our experiments have shown very promising overall performance and speed-up compared to the execution of the original (non-rewritten) query, we estimate that a lot of optimizations are further achievable. These optimizations stand on the fine frontier between theoretical and practical improvements, and involve among others the analysis of the impact on global performance of the order of rules application, an investigation of the locality of changes produced by individual rules, a study of the most suitable data structures. Moreover, while we have seen that the completeness of the rewrite algorithm employing the rules extends in practice beyond the considered theoretical restrictions, we believe it to be of interest to further explore from a theoretical point of view such behaviour. This could also allow establishing a tight tractability frontier for the considered problems.

Last but not least, recall that the soundness and completeness concept in [16] and [56] amounts to finding a rewriting if (at least) one exists. The candidate rewrite plans constructed can be very redundant. While we have shown several practical optimizations to remove redundancy, we note further that these plans do provide all the necessary space for finding all rewritings and in particular the minimal ones. It then becomes interesting to investigate the usage of techniques such as those presented in the first chapter, adapted to an XML setting.

As the title of this thesis suggests, the two chapters approach the problem of view-based rewritings and query reformulation from a theory-oriented as well as practically-driven perspective. In our vision, these two perspectives must be interleaved and constantly enforce one another to achieve meaningful results. We further believe that, although they have long been of general interest in database research, the topics investigated in this work will never run out of new, more complex or larger scale applicative scenarios, thus continuously providing new opportunities and challenges for both theoretical and practical optimization.

Appendix A

Additional topics

We present in the following two additional topics that have been explored during this PhD. Although the topics exposed hereafter diverge from the problem of view-based rewriting, they stay in the broader range of *query acceleration*, being essentially concerned with *indexing strategies*.

A.1 Efficient multi-dimensional indexing (the ACM SIGMOD Programming Contest 2012)

A.1.1 Problem description

The task proposed by the 2012 ACM SIGMOD programming contest (http://wwwdb.inf.tu-dresden.de/sigmod2012contest) is an interesting revisiting of a *classical scenario*, concerning *multi-dimensional indexing*. The contest requirement consists in the design and implementation of a multidimensional, high-throughput, in-memory index structure and its surrounding database layers, such that the resulting system should support common database operations such as point and range queries as well as data manipulation. An important particularity of this problem is the targeted *highly concurrent setting*, comprising many client threads operating queries and updates in parallel.

All index structures and indexed data are required to fit entirely in main memory; a maximum of 32 indexes can be created in parallel, for a data-to-RAM ratio of 10%. The indexes have to support *transactions*, each containing point and range queries (possibly with wildcards) as well as updates (for a maximal update rate of 40%).

The total amount of indexed data is expected to go *up to billions of tuples*, the maximal number of dimensions per tuple being 32, and the type of attributes being either 4 bytes integer values, 8 bytes integer values, or string/VARCHAR (for details, see http://wwwdb.inf.tu-dresden.de/sigmod2012contest/#task-overview). Importantly, the output of range and partial-match point queries have to be *order preserving* (lexicographical byte order for VARCHARs).

A.1.2 Proposed solution

Analysing the problem specification, as well as testing the performance of a draft design, has allowed us to identify two major factors impacting performance: (a) the highly parallel setup

and (b) the “ORDER BY” implicit constraint imposed, i.e. the fact that results are required to be returned according to their global order.

The latter point has a strong consequence on the efficiency of the chosen index structure. Indeed, an important variety of multi-dimensional index structures described in the literature *implicitly impose, by their construction, other types of ordering on the data*. To reconcile the natural retrieval order of these methods with the imposed output order, one would then have to: (a) always search the entire structure, and (b) re-sort the results, hence adding complexity and significantly decreasing performance.

A.1.2.1 Global structures, iterators, transactions

With our solution, we targeted simplicity and robustness. Our global architecture thus uses per-thread data such as transactions, iterators and index handles, as well as "shared" data, that is, indexes themselves. Figure A.1 shows a high level view of these architectural bricks.

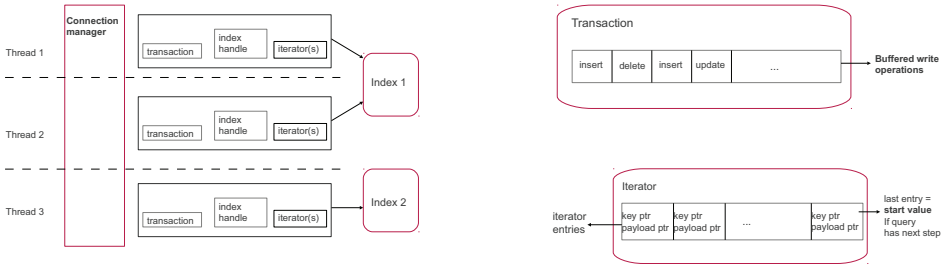


Figure A.1: Architectural bricks

An important design point concerns transaction handling for write operations. Given the isolation level required (read committed) and the highly parallel context, our approach is to: (a) buffer the write operations through the transaction lifecycle and (b) either discard them upon abort or execute them atomically upon transaction commit (note however that this does not forbid granular locking strategies). Conceptually, this induces an order of all transactions on a timeline and makes one transaction’s write execution “instantaneous”, corresponding thus to a point on the time axis.

Another important design choice concerns iterators. First of all, all results stored in the iterator result array are always sorted according to global order. Our design also fixes a maximal bound on the number of results (according to the query type). A query yielding more results than this maximal bound will be executed in several steps - whenever an iterator access beyond that bound occurs, a new search is issued. This new search will however only consider results starting from a minimal value, corresponding to the maximal value of the previous step. Such multi-step execution is in turn enabled by the global ordering that is natively ensured by our design.

One last point concerning iterators is connected to the choice of handling transactions. The iterator is first filled with persistent results from the index; then, before any result is available to

the client, the iterator's contents are adjusted according to the current transaction's write buffer, as these operations should be visible in the current transaction.

Our choices concerning transactions and iterators have a lot of useful consequences. Our approach on transaction handling removes the need for any temporary index copy or rollback management. Fixing a bound on the iterators allows control of queries returning a large number of results; moreover, results are gradually made available in fast increments. We temporarily *stop the search* whenever the iterator is full, while still making the whole result set available if necessary. This behaviour is achieved furthermore in a completely transparent manner from a client point of view.

The drawbacks of our approach are the delay of write operations execution for committed transactions and the lack of adaptiveness for the iterator capacity. We note however that the latter point could be easily managed by adding a "LIMIT" parameter to the provided query API.

A.1.2.2 Core index structure

As underlined above, the global ordering turns out to be of major importance performance-wise. To adapt to this essential requirement, we have thus designed an index structure that *natively* preserves this global ordering, while of course aiming at the best speed and space efficiency.

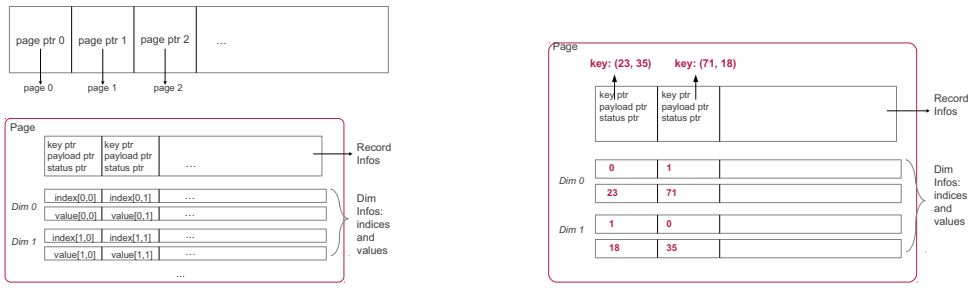
In a nutshell, our core index structure is constructed as follows: data is stored in a globally ordered array, further divided in *pages*, thus providing *a page array*. All data inside one page and among pages is stored and maintained in a manner that respects the global order. The global index structure can thus be seen conceptually as a huge ordered array, and operationally as a two-level tree. Accordingly, we allow for insertion and locking granularity, while avoiding the overhead caused by the depth of a full tree.

On the page level, two different structures are used for handling queries. The first one is the so-called *RecordInfos*, an array storing pointers to original keys and payloads, as well as a status field related to records' visibility (they may be not visible due to deletion or in-process transactions). RecordInfos are of course always sorted according to the global order on keys. The second structure on page level consists in the *DimInfos*, two arrays for each dimension, holding indices in the RecordsInfos array and values on the corresponding dimension. Figure A.2(a) shows the index structure, while Figure A.2(b) details the per-page arrays for a simple example of two records in a two-dimensional setup.

Any insertion in these structures corresponds to a classical sorted array insertion and comprises 3 steps: a new incoming record will be first inserted in the RecordInfos array; then, we update the indices array; as a final step, we insert in both the indices and values array.

Pages have a maximal size. Upon reaching this size, a new page is created and the original full page's contents redispatched between the two, with respect to the ordering imposed on both RecordInfos and DimInfos. Due to the page structure, this split operation is linear and straightforward. The new page is then inserted immediately after the original page in the page array, ensuring that the global order among pages is thus preserved.

Insertions, deletions, updates, and "fully-specified" point queries (that is, queries that have as input a complete key, without wildcards) are handled by binary search on the page array, then on the page level. The sorted DimInfos come in use when handling range and wildcard point



(a) (b)

Figure A.2: Index structure

queries. We first binary search the page array using the value on the first dimension (which corresponds to the global order) and possibly the iterator's first value when dealing with a multi-step query. On the selected pages, we further proceed to a binary search for each dimension by using the values arrays, and fill a page-local bitmask using the indices array. Bits set in the bitmask are those corresponding to results valid on every dimension (a binary *and* operation) and they thus correspond to valid results. The order of the bits set in the bitmask is of course the global order, since it corresponds to the order in the RecordInfos array.

Given the fact that, by the design of our structures, the results are ordered from the very beginning, no reordering is necessary – these results can thus be added gradually to the iterator; moreover, when the iterator reports as full, the search can stop while guaranteeing that *no intermediate data has been missed*. The simple two level structure further ensures locking granularity, thus lowering insertion cost without the overhead that would be induced by cascading splits.

A.1.2.3 Lower-level optimizations

As our extensive evaluation has shown, memory management is particularly important in conjunction with specific data structures, as it can boost them or significantly slow them down. When dealing with a large data set and performance critical implementations, malloc (dlmalloc) is too complex and therefore not adapted, calling for a dedicated allocator.

In the setting we target, we handle a small set of object types and we can benefit from a *simple and specific allocation policy*. In practice, this means that we can for instance choose to dedicate a maximal amount of memory to the application and book this amount by mapping it and then allocating in constant time, by simply moving a cursor. Further, for speeding up allocation and object disposal we can set up *object pools*.

NUMA (*non uniform memory access*: memory is divided into two NUMA nodes corresponding to the two processors; one processor can access its local memory -node- much faster than non-local memory) can be leveraged in a multi-threaded setup by imposing a concept of *locality*. We can achieve this by first *separating memory* for each index (this means no shared memory areas), and then dispatching indexes and their attending threads on matching proces-

sors/nodes, so that a thread would always use its processor's local memory. This strategy can in turn be implemented in a simple round-robin fashion.

Finally, *cache analysis* by relying on tools such as Valgrind/Cachegrind and VTune proved to be very useful for increasing the runtime speed of our system and identifying performance bottlenecks (last level cache misses, split stores, etc.). We note that the indices and values arrays (the DimInfos), although apparently redundant (key contents is copied in the values array), are well adapted to cache usage and therefore turn out to provide very satisfactory performances. Iterating in a predefined order through pages also makes *prefetching* useful (as the next page to be treated is always known in advance).

A.1.3 Conclusions

The contest's setting essentially shows that performance improvement is always achievable, even in the resolution of a classical problem, and further underlines the paramount impact of optimization when dealing with very large amounts of data.

The work presented above has lead us to important insights on the interconnections between theoretical complexity and low-level optimizations, and emphasized the need for efficient data structures that are further designed so as to efficiently fit hardware constraints. Our solution, rewarded with the second prize, proved to be, according to the benchmarking scenarios, 5 to 10 times faster than the reference solution based on Berkeley DB.

A.2 Web source selection for wrapper inference

A.2.1 Structured Web sources and wrapper inference

We are witnessing today the presence of an important amount of structured, "schematized" Web sites. These sites usually publish sets of pages (*sources*) generated dynamically, by means of a formatting template over a database, and the regularity and often uniform typing of the data fields make it possible to develop dedicated techniques for extracting their published data, called *wrapper inference techniques*.

A wrapper's output target is thus a set of complex, possibly hierarchically organized objects, whose attributes are extracted from the input pages. From the early, "manual" approaches, most of the time asking for expert users and the knowledge of dedicated programming languages, through user-friendly interfaces and semi-supervised approaches using various learning techniques, wrapper inference has evolved towards fully automatic approaches.

If we consider the to-be-wrapped pages as the result of formatting data by means of a template, as stated above, the automatic-wrapping process may be seen as an attempt to recover this template, be it explicitly or implicitly. Recent techniques such as those employed by RoadRunner [22], ExAlg [5], or DEPTA [72] make use of the regularity of pages, at the text, HTML encoding/DOM tree and even visual rendering level; they use textual patterns or tree matching algorithms as well as geometrical layout analysis.

The general methodology employed in most recent works consists roughly in two sequential steps, data extraction and data labelling. In practice, this generic approach suffers from shortcomings that often limit the usability of the collected data in real-life scenarios. Thus, without

insight over its content, data resulting from the extraction may mix values corresponding to distinct attributes of the implicit schema, making the subsequent labelling phase tedious and error-prone. Moreover, a lot of "useless" (from a final user point of view) data may be extracted, even if it corresponds to valid attributes of the page objects.

In [25], by means of the ObjectRunner system, the authors address these shortcomings by proposing a wrapping approach that exploits prior knowledge over the data that should be extracted, in what could be seen as targeted wrapping and extraction. In the a first step, of structure specification, users provide an intentional description of the data that is targeted – called a Structured Object Description (SOD). SODs describe nested relational data with multiplicity constraints, starting from atomic types with associated recognizers (such as regular expressions or dictionary-based). This flexible and lightweight initial specification allows discarding "superfluous" data; the dictionary-based atomic type recognizers may be related to a general purpose knowledge base/ontology, thus bringing data semantics earlier in the process of the wrapping.

Extraction is operated by adding data annotations (according to the recognizers) and improving the techniques from [5] (equivalence class definition and refinement as a means of expressing regularity) by the insight given by the information in the SOD, to properly select targeted data. The authors' extensive experimental evaluation of the system thus built shows a significant improvement in accuracy when compared to state-of-the-art wrapper systems.

A.2.2 Selecting relevant sources for wrapper inference

When considering automatic wrapper inference – be it very accurate – in a practical, end-to-end scenario, there is however one major problem complementary to the wrapping strategy: how does one find and select data sources for wrapping? Indeed, several large repositories of crawled Web data are available nowadays. However, applying a general-purpose wrapper on this massive amount of data would mean completely ignoring any "user data need" and wasting a lot of computational resources, be it at the CPU or storage level.

ObjectRunner's targeted extraction paradigm offers a means to discard superfluous data, accelerate the wrapping process and improve accuracy. However, when specifying an SOD for extraction, we implicitly wish to use Web sources conforming to a certain schema, and although ObjectRunner's extraction strategy would theoretically discard unmatching sources, we would still have to examine the whole set of sources available. Moreover, we would have to do this every time we face a new *data need* expressed in the form of an SOD.

To improve performance, we would then benefit from a lightweight (in terms of computational resources consumption) yet effective processing of the available sources, so that when presented with a targeted extraction task (SOD) we could rapidly select the most promising candidates for this task. In doing so we further wish to go beyond traditional text-based indexation techniques, by capturing not only matching content but matching structure as well. In defining this task, we identify the source processing as an "offline" step, that could be coupled with the crawling, whereas source retrieval is to be done "online"; the two performance constraints are therefore different (the online step should be very fast), but we further target global minimization of the execution time and storage requirements.

The source selection problem can then be formally defined as follows: starting from a very large repository of Web sources (sets of pages), given an input SOD s, find the k most relevant

sources for the extraction of instances of s. The main idea of our joint work with Nora Derouiche on this topic is, as sketched above, that of producing a lightweight representation for a given web source – a **source signature** – and further assembling these signatures in an **index**, from which online retrieval would be fast and accurate.

A.2.3 Source signatures

In our first attempt to model this process, we have restricted the SOD's atomic types definition, in order to achieve a more lightweight and general language for the signature. Thus, our technique takes as additional input a general purpose knowledge base (ontology) organized in a hierarchy of concepts and instances thereof, which provides the reference for the dictionary-based recognizers; the query SOD's atomic types are then expressed in terms of concepts from the reference ontology.

Since we deal with HTML pages and complex, nested objects, a natural way of representing source content is by means of a tree. Furthermore, given the assumption of a similar structure across different pages for the structured sources we consider, as well as on page level for pages listing several objects, we would like the tree's hierarchy to reflect the common hierarchy of page *blocks*, such as object fields and objects themselves. In order to map the structure to our search vocabulary, this tree's leaves would bear, instead of text and HTML tags, corresponding instances from the reference ontology, such that the content is "translated" in the query language (that of the SOD). These instances could further have additional features, such as multiplicities or confidence scores.

Our initial approach in building such a tree-like signature, also presented in [24], consists in applying a decomposition in visual blocks on a sample of pages from the source using the visual segmentation algorithm of VIPS [15]. We further reduce this visual tree by applying a radical simplification in order to retain the data rich page segment, by heuristically choosing as the relevant page tree root the first level child with the largest and most central rectangle. The resulting visual trees (one for each page of the sample) are then merged using a range of heuristics based on the HTML *id* and *class* attributes, as well as rectangle sizes. The merge process is done exclusively on leaves; for the set of leaves identified as belonging to the same class according to these heuristics, we assign the first VIPS Dewey Id encountered (in the order of sample pages, and for each page in its own induced order). Leaf classes that have the same VIPS Dewey Id are then further merged.

The VIPS Dewey Ids on the classes of leaves induce a virtual tree-like structure, where there is an implicit parent-child relationship; this tree's nodes correspond to the blocks in our definition of signature, and we call the initial leaves before merging *block instances*. Figure A.3[24] shows a schematic representation of the tree for a given source.

Deriving a common structure for the page blocks is a non-trivial problem, closely related to the common workflow of modern wrapping approaches, in the steps of data-rich region identification and record segmentation. At the same time, our approach aimes at avoiding the costly process of wrapper inference, and could thus stop at a "coarser" level, tolerating a higher error rate in terms of page segmentation. Reversely, while our initial approach relies on the visual segmentation produced by VIPS, VIPS's granularity parameter (the Degree of Coherence) tunrs out in some cases to be insufficient for our needs of tree decomposition, thus potentially calling

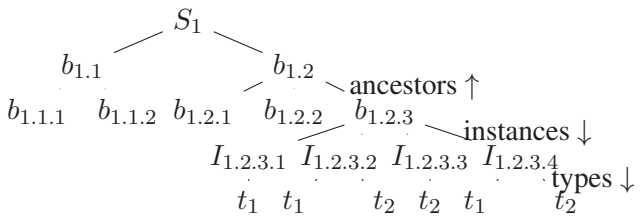


Figure A.3: Source signature: blocks, instances, types

for "light versions" of other segmentatio algorithms, such as DEPTA[72]'s record identification technique.

A.2.4 Semantic annotation and scoring

In our initial approach, all pieces of text between tags in the leaf instances are matched against the input ontology's instances. However, types/categories in an ontology have a lot of common, identically-spelled instances (e.g. "Madonna"), so a natural further step would be a disambiguation one, allowing us to better assign a type to specific object fields. To this end, for a given leaf block instance, we can merge together all annotations occurring at the same relative tag path (starting from the assumption that they represent the same object attribute). We then count for each such tag path the occurrences of the various types and we use a mix of WordNet similarity measures between types to end with the final best type assignment per tag path:

$$bestType(i) = arg\max_{c \in O} \sum_{c' \in O} score(i, c') \cdot T_{sim}(c, c') \tag{A.1}$$

Then, the only type retained in the instance for a given tag path will be the best scoring type. Thus, our leaf instances will be bags of pairs (tag path, associated type). We then mutualize instance information for a given block, while still keeping the type-assignment per-tag-path, so that we end with each block represented as a set of tag-paths, each having an associated count for a given type. We then assign to each type in a block a classic tf-idf score, at the granularity level of blocks.

$$sc(t, b) = tf(t, b) \cdot idf(t) \tag{A.2}$$

where, in order to intuitively counterbalance segmentation errors (slightly different relative tag paths) and avoid mixing object fields, *term frequency* tf is computed cosidering the dominant tag-path for a given type:

$$tf(t, b) = \frac{\max_{path}\{count(t, b, path)\}}{\sum_{t' \in T} \max_{path}\{count(t', b, path)\}} \tag{A.3}$$

The semantic annotation of data is also a challenging matter, raising a range of sub-problems. One obvious issue, which can be addressed by employing offline indexes such as Patricia tries,

is the important quantity of ontology data, which we need to be able to search fast to keep our overall performance target. A promising direction for reducing the search space is to perform "best domain(s)" detection for each source, supposing we deal with ontologies structured by domains. Domain detection can also improve the disambiguation of text matching instances that appear in several categories and thus implicitly improve the annotation accuracy. Regarding the variation in spelling of ontology instances, basic, light NLP processing (e.g. stemming) could also be beneficial in a number of cases; however all these steps have to be carefully chosen in order to keep the fine balance of better accuracy versus higher cost.

Merging block instance semantic information is in our initial approach done only as a final step. Bringing together instance annotations for the same block would possibly provide a better manner to designate best types for a given tag path. At the same time, delaying this merge could provide the opportunity of deriving common structure by taking into account annotations as wellm in addition to visual layout structure.

The scoring strategy is also related to the above issues, raising the question of whether one should consider the significance of a type at the instance, block or even source level; an additional question is related to the choice of a scoring strategy that would allow decreasing the importance of the useless data (e.g. page content that has been wrongly identified as belonging to the data-rich region).

A.2.5 Signature indexing

By encoding the source content in the above general *signature* tree-like form, using the reference ontology language, we can benefit from techniques of top-k keyword search (since we have reduced our content vocabulary to ontology types). The simplest approach would be to consider the set of distinct atomic types in the SOD and do a full top-k keyword search (TA, NRA) on an inverted index containing the annotated leaf blocks.

However, we would like to go one step further and capture the tree-like structure of the published objects and the input SOD. In doing so, one possible approach is to adopt the techniques described in [19] for top-k keyword search in XML trees, which also consider intermediate (LCA) blocks. Indeed, this algorithm matches our setting (scored keywords in tree nodes) but also considers a score on parent nodes by means of child scores aggregation and the usage of a damping function.

The choice of an index for source retrieval is connected to all the other steps of the designed solution, as the performance for a given index structure depends heavily on the segmentation, annotation and scoring performed. A basic inverted index would intuitively provide more accurate answers if the granularity of the deduced blocks is coarser than that of the input SOD (in a way, if leaf blocks encompass data objects, so that all queried types are grouped together in the leaves). On the other hand, an LCA-aware index with aggregation and damping functions would better capture hierarchy and allow for a more flexible block segmentation; however, levels in this hierarchy may be purely syntactical (HTML) and unrelated to the searched object's structure, thus intuitively unsuited for the application of certain damping functions; on the other hand, useless data's contribution may be (by the same kind of mechanism) artificially increased. These latter issues are also related to the targeted data object's representation and the nesting and multiplicities present in the SOD; if we go beyond the representation of the query as a bag

of keywords, we then face a range of schema-matching problems that constitute in themselves a complex topic.

A.2.6 Conclusions

The problem of structured Web sources indexing and selection proves to be a challenging topic. While our initial system jointly developed with Nora Derouiche, built using the naive choices listed previously, shows encouraging performances, in the above we underline a few of the many possibilities of improving speedup and accuracy that are worth exploring in an optimized implementation and empirical evaluation.

Assessing the performance and accuracy of such a system can in turn follow several axis. Retrieval accuracy could be indirectly measured by running a wrapping system over the selected sources, and matching its output against the query SOD. Speed can be measured by comparison with wrapping techniques but also with indexing platforms such as Apache's Lucene. Finally, obtaining relevant massive datasets for testing purposes, a non-trivial issue on its own, is further planned to be achieved by operating a deployment and integration with the large Web archiving platform of the Internet Memory Foundation.

Annexe B

Condensé de la thèse en français

B.1 Introduction

B.1.1 Recherche des données et accès aux données : une perspective pragmatique

Notre époque est caractérisée par l'abondance des données : données personnelles, données d'entreprise, données générées par des capteurs, quantités massives de données venant du Web et des réseaux sociaux. Le terme *big data* fait maintenant partie du vocabulaire courant. Grâce aux capacités toujours croissantes de stockage et calcul, ces *big data* semblent toujours plus faciles à traiter, et il y a une tendance croissante à ne pas prendre en compte les ressources impliquées dans la recherche et l'accès à ces données.

Le coût est pourtant là. D'un point de vue purement financier, ce coût peut être vu comme le prix prohibitif de l'équipement de stockage et calcul, ou des frais d'accès à des services *cloud* si le calcul et le stockage sont externalisés. Dès que les ressources sont limitées par des considérations financières, ce coût devient visible en tant que manque de performance dans la recherche et l'accès aux données : soudain, on est face à 15 minutes d'attente pour l'exécution d'un requête SQL de taille moyenne sur une base de données de taille plus que raisonnable. Cela peut fortement surprendre l'utilisateur de Google habitué à une réponse instantanée à sa requête sur le vaste World Wide Web.

C'est typiquement dans ces situations que le besoin d'optimisation dans la recherche et l'accès aux données revient au premier plan. On devient conscient de la nécessité d'avoir des algorithmes performants qui permettent de diminuer le stockage et le transfert et de rendre les recherches plus rapides, sans impliquer des ressources additionnelles. Les *accélérateurs* de la recherche et de l'accès comme les caches, les vues matérialisées et les indexes retrouvent leur intérêt, après avoir été négligés à cause de la fausse certitude que la recherche et l'accès à n'importe quelles données soient intrinsèquement rapides. Toute opportunité d'améliorer les performances pratiques devient un but désiré : un traitement efficace en mémoire principale, des algorithmes polynomiaux ou même *moins exponentiels*, des implémentations adaptées et optimisées.

B.1.2 Les vues matérialisées : un moyen d'améliorer la recherche et l'accès aux données

Parmi les accélérateurs de la recherche et de l'accès, les vues matérialisées et les caches sont depuis longtemps connus pour leur capacité à améliorer les performances des requêtes. Alors que le terme *vues* a une connotation liée aux bases de données, *cache* est de nos jours un terme omniprésent, par exemple dans les clients et serveurs Web. Les deux termes expriment fondamentalement la notion de court-circuiter une opération coûteuse d'accès distant et/ou de calcul, nécessaire dans la recherche ou l'accès aux données, en matérialisant (potentiellement de façon locale) des résultats pré-calculés.

L'utilisation des vues pour baisser le coût de la recherche et de l'accès aux données génère en revanche un ensemble de problèmes complexes, comme par exemple la question de savoir quelles sont les vues qu'il faudrait matérialiser pour maximiser l'efficacité de l'accès, et comment ces vues doivent être maintenues à jour de façon efficace. Aussi, pour obtenir un gain de performance en utilisant des vues matérialisées, le coût de leur sélection et maintenance doit être largement contrebalancé par l'accélération obtenue en les employant dans la recherche et l'accès aux données. Supposant tous ces problèmes résolus, il reste la question essentielle de savoir si et comment des vues existantes peuvent être utilisées pour répondre à une requête donnée : autrement dit, le problème de la *réécriture de requêtes avec des vues*.

B.1.3 Réécriture de requêtes avec des vues et reformulation de requêtes

Outre les scénarios classiques d'optimisation, dont le but est d'accélérer l'exécution d'une requête en se basant sur des vues matérialisées, la réécriture de requêtes avec des vues peut aussi être placée dans le cadre général de la *reformulation de requêtes* : étant donnée une requête Q formulée sur un schéma source S, trouver une requête équivalente R exprimée par rapport à un schéma cible T, en exploitant la relation entre S et T. La reformulation de requêtes comprend de nombreux autres problèmes qui ont préoccupé la recherche en bases de données pendant des décennies, comme la sélection de chemin d'accès physique et l'optimisation sémantique (élimination des jointures redondantes et autres instances de réécriture de requêtes sous des contraintes d'intégrité).

Les vues peuvent ainsi être considérées non seulement comme des accélérateurs, mais aussi de façon plus générale comme des *modèles de l'accès aux données*. Par example, les vues peuvent être utilisées pour exprimer des points d'accès sécurisés dans un contexte de restriction d'accès. Dans ce cas, l'accès par les vues n'est pas ciblé pour le gain potentiel en performance, mais essentiellement parce qu'il constitue *le seul accès possible*. Un scénario similaire, mais plus complexe, est celui du *prix associé aux données* ; dans ce cas, l'accès n'est pas uniquement restreint, il comporte aussi un prix diffèrent en fonction des vues employées. Les systèmes de type médiateur et les architecture multi-stockage et multi-modèle peuvent aussi être décrits avec des vues, fournissant ainsi une variété d'occurrences pratiques du problème de reformulation de requêtes et de réécriture de requêtes avec des vues.

B.1.4 Structure du manuscrit et contributions

Dans ce document, nous adressons le problème de la réécriture de requêtes avec des vues, en adoptant une perspective en égale mesure théorique et pratique. Nous accordons un poids important à l'analyse théorique, à la correction et à la complexité ; en même temps, nous gardons constamment une démarche pragmatique, et une partie importante de nos développements théoriques sont motivés par le besoin d'atteindre l'efficacité sur un plan pratique..

Dans le **premier et principal chapitre** de cette thèse, nous approchons le sujet de la **recherche de reformulations minimales conjonctives pour des requêtes relationnelles conjonctives, sous des contraintes d'intégrité**, qui incluent (mais ne sont pas limitées à) la relation entre les schémas source et cible. Une reformulation est dite minimale si elle ne contient pas dans sa clause FROM des éléments redondants, qui ne sont pas nécessaires pour assurer son équivalence avec la requête d'origine, sous les contraintes données.

Tous les algorithmes de reformulations auxquels nous nous intéressons dans ce travail doivent être *corrects*, c'est-à-dire de retourner des reformulations valides. Dans le premier chapitre, nous accordons aussi une importance majeure au concept de *complétude*. De façon générale, la complétude (ou *complétude forte*) d'un algorithme de reformulation, par rapport à une classe de solutions, désigne sa capacité à trouver *toutes* les reformulations dans cette classe. L'intérêt immédiat et central de trouver toutes les reformulations minimales est le fait que, pour une vaste majorité des modèles de coût, les reformulations de *coût minimum* seront toujours parmi les reformulations minimales.

La complétude est ainsi clairement souhaitable dans les scénarios pratiques qui définissent la mesure d'une requête comme le coût minimum de toutes ses reformulations. Considérons par exemple le renforcement du contrôle d'accès par des *vues de sécurité* [54, 61], où une requête est autorisée uniquement si elle a une réécriture qui utilise un ensemble de vues autorisées. Supposons un raffinement de ce scénario, où l'accès à chaque vue demande un niveau d'autorisation, et où un analyste a besoin de connaître, pour le demander auprès des responsables, le niveau minimum d'autorisation nécessaire pour exécuter une requête. Ceci implique de trouver toutes les réécritures de la requête en question, et de sélectionner parmi elles celle qui demande le niveau minimum d'autorisation. Le même type de raisonnement peut être appliqué dans les scénarios de *prix associé aux données* [43], où le propriétaire des données fixe le prix pour un ensemble de vues sur ses données. Le coût d'une requête peut par la suite être établi de façon automatique comme le prix minimum des reformulations possibles. La complétude est aussi essentielle dans le cadre de l'optimisation classique de requêtes, car la meilleure reformulation parmi celles trouvées par un algorithme incomplet peut être de façon significative moins rapide à l'exécution que celle de coût optimum, qui est garantie par un algorithme complet. En effet, comme le montre notre évaluation expérimentale, le temps d'exécution de la meilleure reformulation trouvée par un optimiseur sophistiqué dans un SGBD commercial peut être jusqu'à deux ordres de magnitude plus élevé que celui nécessaire pour l'exécution de la reformulation de coût minimum.

Toutefois, étant donné que dans le cas particulier du problème de reformulation qui correspond à la réécriture totale avec des vues d'une requête, le problème de décision associé est NP-difficile même en l'absence des contraintes [46], le point de vue général jusqu'à présent a

été celui de considérer la complétude comme un concept d'intérêt purement théorique. En effet, pour l'algorithme Chase & Backchase [27], à notre connaissance le seule algorithme complet dans ce contexte, la recherche de reformulations minimales ne peut pas passer à l'échelle en dépassant le spectre inférieur des tailles de requêtes et contraintes rencontrées en pratique. Ceci est dû au fait que, même si très peu de reformulations minimales existent, le Chase & Backchase inspecte un nombre de candidats qui est souvent exponentiel dans la taille de la requête et le nombre de vues, lançant ainsi un nombre exponentiel de séquences de *chase*. [60] confirme ce fait de manière expérimentale, puis consacre la majeure partie de ses résultats à des approches heuristiques qui réduisent l'espace de recherche en sacrifiant la complétude pour gagner en performance. Des démarches similaires sont adoptées par toutes les implémentations existantes de la recherche de reformulations, y inclus les optimiseurs des SGBD et les implémentations basées sur le Chase & Backchase pour trouver les reformulations de requêtes XML [29, 57, 70].

Dans ce travail, nous remettons en question les opinions précédentes sur l'incompatibilité entre la complétude et la performance, en présentant un **nouvel algorithme correct et complet**, le Provenance-Aware Chase & Backchase, qui résout le problème des reformulations minimales avec **des performances significatives sur le plan pratique**. Nous fournissons **sa caractérisation théorique détaillée et son implémentation**. Nous présentons par la suite son **évaluation expérimentale**, montrant des gains de performance jusqu'à deux ordres de magnitude entre l'exécution d'une reformulation optimale trouvée par un SGBD commercial et de celle trouvée par notre algorithme (que le SGBD manque de trouver avec son algorithme *incomplet*). Nous montrons ensuite comment adapter notre algorithme pour **trouver directement des reformulation de coût minimum**, pour des fonctions de coût monotones, et les améliorations supplémentaires de performance que cette adaptation rend possibles.

Le Provenance-Aware Chase & Backchase transforme le Chase & Backchase en employant une technique beaucoup plus ciblée de recherche de reformulations. La raison principale de la performance atteinte par le Provenance-Aware Chase & Backchase est le fait que le nombre de séquences de *chase* potentiellement exponentiel dans le Chase & Backchase est remplacé dans notre algorithme par *une seule telle séquence*, employant **une nouvelle technique de chase**, la Provenance-Aware Chase. Comme son nom l'indique, la Provenance-Aware Chase est une procédure de chase qui utilise des informations de *provenance*, permettant de retrouver les reformulations minimales recherchées. Le type de provenance utilisé correspond à la *minimal-why provenance*, introduite dans un but différent dans [14]. La conception de la Provenance-Aware Chase a été complexe et difficile sur le plan théorique. En effet, la technique standard de *chase* n'est pas compatible avec les annotations de provenance, créant le besoin de conception d'une technique additionnelle, non-annotée, qui puisse respecter les propriétés désirées, et que nous appelons la Conservative Chase. Dans sa description en tant que la Conservative Chase instrumentée avec de la provenance, outre son usage dans notre algorithme de reformulation, la Provenance-Aware Chase devient intéressante en soi, comme moyen de **raisonnement sur l'interaction entre la provenance et les contraintes**.

Dans le **deuxième chapitre** de cette thèse nous nous plaçons dans un contexte XML et nous revisitons le travail précédent de Cautis, Deutsch et Onose, présenté en [16] et détaillé en [56], sur le sujet des **réécritures de requêtes XPath avec des vues**. Le type de réécritures analysées

comprend **un seul niveau d'intersection de plusieurs vues** : il s'agit donc de réécritures comprenant de la navigation dans les vues, une intersection, et potentiellement une dernière étape de navigation supplémentaire. Le travail que nous revisitons présente une analyse de complexité du problème et un algorithme correct et complet pour sa résolution. Comparé au contexte analysé dans le premier chapitre, le concept de complétude ciblé ici est celui de *faible complétude* : un algorithme est dit complet dans ce cas s'il trouve *au moins une reformulation* dans une classe donnée C dans tous les cas où (au moins) une telle reformulation existe. On peut remarquer qu'un algorithme (faiblement) complet de reformulation peut servir en tant que procédé de décision pour l'existence d'une reformulation en C, tout en étant *plus riche* car il doit être aussi capable de retourner la reformulation. Ce comportement peut être très utile en pratique, par exemple dans les scénarios d'accès restreint par des *vues de sécurité* mentionnés précédemment (dans leur version initiale, sans niveau d'autorisation associé). L'accès par les vues étant le seul accès possible, il est essentiel dans ce contexte de trouver une reformulation dès qu'une reformulation (c'est-à-dire, un accès possible) existe.

Notre principale motivation pour le travail présenté dans le deuxième chapitre est celle de trouver et appliquer des stratégies pour atteindre des performances significatives sur le plan pratique. Suite à l'analyse de complexité montrant la difficulté du problème, [16] présente et [56] détaille l'utilisation d'une technique à base de règles pour inférer un algorithme plus rapide mais uniquement *correct* pour le problème de réécriture, ainsi que des conditions pour que cet algorithme devienne complet. Nous **raffinons** la technique à base de règles pour assurer sa **complexité polynomiale** et **améliorons les conditions de complétude** de l'algorithme de réécriture correspondant. Nous présentons ensuite **un ensemble d'optimisations** des techniques de réécriture, nécessaires pour atteindre des **performances pratiques**. Nous fournissons **une implémentation complète** des techniques de réécriture, comprenant les optimisations et raffinements proposés, et présentons son **évaluation expérimentale** extensive, montrant ses performances et son utilité.

Notre investigation du travail en [16] et [56] a aussi une conséquence importante sur le plan théorique : nous **enrichissons l'analyse du problème de réécriture** en montrant, structurant et clarifiant ses connections avec le problème de décision de l'équivalence entre une requête exprimée par un *DAG pattern* et une requête exprimée par un *tree pattern*, et avec le problème de la *union-freeness* d'un *DAG pattern*, c'est-à-dire de trouver une requête *tree pattern* équivalente à une requête *DAG pattern*.

Le premier chapitre de cette thèse étend notre article [42] : **Ileana, Cautis, Deutsch, Katsis, *Complete yet practical search for minimal query reformulations under constraints***, SIGMOD Conference 2014, 1015-1026.

Notre raffinement de $Prov_{C\&B}$ pour trouver directement les reformulations de coût minimum, présenté dans le premier chapitre, est au cœur du système ESTOCADA, présenté dans le papier (couramment sous revue pour CIDR 2015) : **Bugiotti, Bursztyn, Deutsch, Ileana, Manolescu, *Invisible Glue : Scalable Self-Tuning Multi-Stores***.

Enfin, le deuxième chapitre étend notre contribution à l'article de journal (couramment sous revue pour TCS) : **Cautis, Deutsch, Ileana, Onose : *Rewriting XPath queries using view intersections : tractability versus completeness***.

B.1.5 Autres sujets explorés par ce travail de thèse

Alors que ce manuscrit se focalise sur le problème des réécritures de requêtes avec des vues, le travail de cette thèse comprend également d'autres sujets, appartenant à la gamme plus large des accélérateurs de requêtes.

Les deux principaux tel sujets, explorés en détail, et présentés dans l'Annexe A, sont liés à l'*indexation*. Le premier sujet, fourni par le **Concours de Programmation ACM SIGMOD 2012**, concerne la conception d'**une structure d'index multi-dimensionnelle**, efficace et stockée en mémoire principale, qui puisse répondre à des requêtes de type point ou intervalle, ainsi que traiter des modifications de données, dans un contexte **fortement concurrentiel**, consistant en de nombreux *threads* client qui effectuent des requêtes et des modifications en parallèle. Nous présentons dans la Section A.1 notre travail sur ce sujet, qui a été récompensé par le **second prix dans le concours**.

Le deuxième sujet lié à l'indexation concerne **l'indexation et la sélection de sources Web structurées** pour l'inférence de *wrappers*. Les sources Web structurées sont des ensembles de pages Web qui ont un contenu structuré similaire, comme par exemple les pages des livres sur Amazon.com. Le *Web wrapping* consiste en l'extraction des données de ces pages, en s'appuyant sur leur similarité structurelle. La sélection de sources suppose une description sommaire, fournie par l'utilisateur, du type de données qu'on souhaite extraire, et l'usage de cette description pour sélectionner, en passant par une structure d'index, parmi les sources Web préalablement parcourues et indexées, celles qui publient le type de données requis. Nous présentons dans la Section A.2 notre travail sur ce sujet, qui étend le travail précédent de Derouiche, Cautis et Abdelssalem, et qui a été effectué dans le cadre du projet Arcomem.

Finalement, un troisième sujet exploré est un sujet qui se situe au croisement des stratégies d'indexation et des réécritures avec des vues. Ce sujet concerne le problème de l'**indexation de vues**, ayant comme but l'accélération du calcul de réécritures. Alors que notre étude de ce sujet en est encore à ses debuts, nous considérons cette approche particulièrement intéressante à poursuivre dans un futur travail de recherche, en tant que moyen de fournir des gains complémentaires en performances pour les stratégies de réécriture présentées dans ce manuscrit.

B.2 Condensé du premier chapitre

Dans le premier chapitre de cette thèse, nous présentons l'algorithme *Provenance-Aware Chase & Backchase* ($Prov_{C\&B}$), pour trouver des reformulation minimales conjonctives pour des requêtes relationnelles conjonctives, sous des contraintes d'intégrité. Le $Prov_{C\&B}$ transforme l'algorithme classique de Chase & Backchase ($C\&B$) [27] dans un but simple et clair : *préserver la complétude* (une propriété essentielle du $C\&B$), mais *atteindre des performances significatives sur le plan pratique* (que le $C\&B$ manque de fournir).

B.2.1 Rappel de l'algorithme Chase & Backchase

Le $C\&B$ est un algorithme qui trouve toutes les reformulations minimales conjonctives pour une requête conjonctive, sous des contraintes d'intégrité qui incluent la relation entre le schéma source S et le schéma cible T. Cet algorithme se remarque par sa *complétude*, c'est-à-dire sa

capacité à trouver *toutes les reformulations minimales*. Les contraintes traitées par le $C\&B$ couvrent la gamme des *embedded dependencies* [1], comprenant donc les *TGDs* (*tuple generating dependencies*) et les *EGDs* (*equality generating dependencies*). Nous allons présenter le fonctionnement du $C\&B$ en montrant son comportement sur un exemple simple, comprenant la description d'un problème de *réécriture totale avec des vues*, où on est donc intéressé à trouver toutes les réécritures minimales qui utilisent *uniquement les tables correspondant aux vues*.

Exemple B.2.1. *Imaginons qu'un éditeur de logiciels représente une partie de ses données internes par le schéma suivant :*

R(A,B,C), S(C,D), T(D,E).

La table R table montre l'appartenance des ingénieurs logiciel à des équipes, en tant que tuples id ingénieur(A), rôle ingénieur(B), id équipe (C). Un ingénieur peut participer dans plusieurs équipes et peut potentiellement tenir plusieurs rôles au sein d'une même équipe. La table S représente la participation des équipes sur les produits, en tant que tuples id équipe (C), id produit (D). Une équipe peut bien sur travailler sur plusieurs produits, et plusieurs équipes peuvent collaborer sur un produit donné. Enfin, la table T table énumère les incidents de production de haute priorité, en tant que tuples id produit (D), id incident(E).

Pour une résolution rapide des incidents, le responsable QA doit envoyer des e-mails à tous les ingénieurs qui pourraient aider dans la résolution de ces incidents. La liste de ces ingénieurs peut être obtenue par la requête suivante[1] *:*

.

Q : select r.A from R r, S s, T t where r.C=s.C and s.D=t.D,

Supposons maintenant l'existence des vues matérialisées :

V_R(A,C) : select r.A, r.C from R r
V_S(C,D) : select s.C, s.D from S s
V_{RS}(A,D) : select r.A, s.D from R r, S s where r.C=s.C
V_T(D,E) : select t.D, t.E from T t

V_R montre la participation des ingénieurs dans les équipes (sans tenir compte de leur rôle). V_{RS} montre la participation des ingénieurs sur les produits. Il est facile de vérifier que les requêtes suivantes :

R_1 : select v_r.A from V_R v_r, V_S v_s, V_T v_t where v_r.C=v_s.C and v_s.D=v_t.D
R_2 : select v_{rs}.A from V_{RS} v_{rs}, V_T v_t where v_{rs}.D=v_t.D

sont des réécritures équivalentes de Q avec les vues données (ce sont des réécritures totales*, qui utilisent* uniquement *les vues). Ce sont aussi des réécritures minimales, et elles constituent l'ensemble des réécritures minimales possibles.*

[1]Toutes les requêtes dans ce travail ont une sémantique *set*, donc le SELECT est un SELECT DISTINCT ; on omet le mot DISTINCT pour des raisons de concision.

Le $C\&B$ analyse ce problème comme un problème de reformulation où le schéma source est celui de la requête Q (tables R, S, et T) et le schéma cible est celui des vues matérialisées (tables V_R, V_S, V_{RS} et V_T) – on rappelle qu'il s'agit d'un problème de réécriture totale avec des vues. Dans ce cas simple, il n'existe pas de contraintes supplémentaires à part celles qui relient les deux schémas. L'ensemble de contraintes $\mathcal{C}$ reliant les deux schémas est obtenu à partir de la définition des vues comme suit :

$$
\begin{array}{ll}
c_{V_R} : \forall r, r \in R & \rightarrow \exists v_r, v_r \in V_R \wedge v_r.A = r.A \wedge v_r.C = r.C \\
b_{V_R} : \forall v_r, v_r \in V_R & \rightarrow \exists r, r \in R \wedge r.A = v_r.A \wedge r.C = v_r.C \\
c_{V_S} : \forall s, s \in S & \rightarrow \exists v_s, v_s \in V_S \wedge v_s.C = s.C \wedge v_s.D = s.D \\
b_{V_S} : \forall v_s, v_s \in V_S & \rightarrow \exists s, s \in S \wedge s.C = v_s.C \wedge s.D = v_s.D \\
c_{V_{RS}} : \forall r, s,\ r \in R \wedge\ s \in S \wedge\ r.C = s.C & \rightarrow \exists v_{rs}, v_{rs} \in V_{RS} \wedge v_{rs}.A = r.A \wedge v_{rs}.D = s.D \\
b_{V_{RS}} : \forall v_{rs}, v_{rs} \in V_{RS} & \rightarrow \exists r, s,\ r \in R \wedge s \in S \wedge r.A = v_{rs}.A \\
 & \quad \wedge s.D = v_{rs}.D \wedge r.C = s.C \\
c_{V_T} : \forall t, t \in T & \rightarrow \exists v_t, v_t \in V_T \wedge v_t.D = t.D \wedge v_t.E = t.E \\
b_{V_T} : \forall v_t, v_t \in V_T & \rightarrow \exists t, t \in T \wedge t.D = v_t.D \wedge t.E = v_t.E
\end{array}
$$

A chaque définition de vue on associe deux contraintes : une depuis le schéma source vers le schéma cible (indiquée par la lettre c), et une deuxième dans le sens inverse (indiquée par la lettre b).

Le $C\&B$ est basé sur la procédure de *chase* [1], qui ajoute à une requête les éléments impliqués par les contraintes. Ceci est réalisé par l'application répétée d'une transformation syntaxique appelée *chase step*. Un *chase step* va chercher un *mapping* de la prémisse (la partie gauche d'une contrainte) dans la requête, et va enrichir la requête pour assurer l'existence par la suite d'une *extension de ce mapping* depuis la conclusion (la partie droite de la contrainte) vers la requête ainsi résultante. Un *chase step* ne s'applique pas si une telle extension existe déjà. Le résultat (unique jusqu'à équivalence homomorphique) d'une séquence complète de *chase* (c'est-à-dire une application répétée de *chase steps* jusqu'à ce qu'aucun *chase step* ne s'applique plus) sur une requête Q avec un ensemble de contraintes $\mathcal{C}$ est noté $Q^{\mathcal{C}}$. Le $C\&B$ comporte deux étapes :

1. **Chase :** On applique une séquence complète de *chase* à Q et on construit le *plan universel* en limitant $Q^{\mathcal{C}}$ au schéma T. Le plan universel a la propriété essentielle de fournir l'intégralité de l'espace de recherche pour les reformulations minimales : en effet, il a été montre que *toutes les reformulations minimales sont isomorphiques à des sous-requêtes du plan universel*.

2. **Backchase :** On examine toutes les sous-requêtes du plan universel U pour savoir si elles sont équivalentes à Q. La vérification d'équivalence est réalisée en appliquant une séquence complète de *chase* à chaque sous-requête et en cherchant ensuite un *containment mapping* [1] de Q au résultat de la séquence de *chase*. Si un tel *containment mapping* existe et la sous-requête est en plus minimale, alors elle fait partie des résultats retournés par le $C\&B$.

Exemple B.2.2. *Continuant avec notre exemple, l'étape de chase consiste d'abord à appliquer une séquence complète de chase sur Q, qui donne le résultat :*

$Q^{\mathcal{C}}$: *select r.A*
from R r, S s, T t, $V_R\ v_r$, $V_S\ v_s$, $V_T\ v_t$, $V_{RS}\ v_{rs}$
where r.C=s.C and s.D=t.D and v_r*.A=r.A and* v_r*.C = r.C and* v_s*.C=s.C*
and v_s*.D=s.D and* v_t*.D=t.D and* v_t*.E=t.E and* v_{rs}*.A=r.A and* v_{rs}*.D=s.D*

Le plan universel résultant (en ne gardant que les éléments de la clause FROM correspondant aux vues, donc au schéma cible) est le suivant :

U : *select* v_r*.A*
from $V_R\ v_r$, $V_S\ v_s$, $V_T\ v_t$, $V_{RS}\ v_{rs}$
where v_r*.C=*v_s*.C and* v_s*.D=*v_t*.D and* v_r*.A=*v_{rs}*.A and* v_s*.D=*v_{rs}*.D*

Dans la phase de backchase, toutes les sous-requêtes de U *sont analysées pour déterminer leur équivalence à* Q*. Ceci consiste en une séquence complète de chase suivie de la recherche d'un containment mapping. Pour* R_2*, le résultat de la séquence complète de chase est :*

$R_2^{\mathcal{C}}$: *select* v_{rs}*.A*
from $V_{RS}\ v_{rs}$, $V_T\ v_t$, *R r, S s, T t*
where v_{rs}*.D=*v_t*.D and r.A=*v_{rs}*.A and s.D=*v_{rs}*.D and s.C= r.C and t.D=*v_t*.D and t.E=*v_t*.E*

La fonction identité étant un containment mapping de Q *vers* $R_2^{\mathcal{C}}$*, la sous-requête* R_2 *est identifiée comme une réécriture. Le* $C\&B$ *vérifie qu'il s'agit aussi d'une réécriture minimale et, ceci étant le cas, ajoute* R_2 *dans la liste de ses résultats. La réécriture* R_1 *est découverte de la même façon, et les autres sous-requêtes sont analysées puis rejetées car l'équivalence n'est pas vérifiée.*

Performance pratique du $C\&B$. La première implémentation du $C\&B$ est décrite dans [60], et la phase de backchase est identifiée comme problématique pour les performances, *en raison du nombre potentiellement exponentiel de séquences complètes de chase (coûteuses) qui sont lancées*. En effet, comme on a vu, ces séquences doivent être appliquées sur chaque sous-requête du plan universel.

La seule amélioration pratique identifiée *qui soit capable de préserver la complétude de l'algorithme* consiste à parcourir les sous-requêtes *dans un ordre croissant de leur taille*. On évite ainsi d'examiner les sous-requêtes dont une sous-requête propre est déjà établie comme réécriture - ces sous-requêtes dont une partie est déjà identifiée comme résultat ne seront pas minimales.

Pour optimiser le cas où il n'existe aucune reformulation, cas dans lequel la stratégie ci-dessus n'améliore en rien les performances, [60] propose en plus de vérifier si le plan universel est lui même une reformulation. En effet, il a été montre que des reformulations peuvent exister si et seulement si le plan universel lui-même est une reformulation. La vérification du plan universel s'effectue de la même façon que pour les sous-requêtes : on y applique d'abord une séquence complète de *chase*, puis on recherche un *containment mapping* de Q vers le résultat de cette séquence.

Exemple B.2.3. *Continuant avec les exemples précédents, une séquence de chase sur U avec $b_{V_{RS}}$, b_{V_R}, b_{V_S} et b_{V_T} donnera :*

$U^{\mathcal{C}}$ *: select* v_r.A
from V_R v_r, V_S v_s, V_T v_t, V_{RS}, v_{rs}, R r_1, S s_1, R r_2, S s_2, T t
where v_r.C=v_s.C *and* v_s.D=v_t.D *and* v_r.A=v_{rs}.A *and* v_s.D=v_{rs}.D *and* r_1.A=v_{rs}.A
and s_1.D=v_{rs}.D *and* r_1.C=s_1.C *and* r_2.A=v_r.A *and* r_2.C=v_r.C
and s_2.C=v_s.C *and* s_2.D=v_s.D *and* t.D = v_t.D *and* t.E = v_t.E

Il existe deux containment mappings de Q vers $U^{\mathcal{C}}$, donc au moins une reformulation existe, et l'analyse des sous-requêtes peut commencer.

B.2.2 Un nouvel algorithme : Provenance-Aware Chase & Backchase

Les optimisations du $C\&B$ présentées ci-dessus, seules à pouvoir préserver sa complétude, ne suffisent pas, malgré leur rôle positif certain, pour assurer des performances pratiques pour le $C\&B$ ainsi raffiné. En effet, il arrive très souvent en pratique que, malgré ces optimisations, le $C\&B$ aie à *analyser un nombre restant exponentiel de sous-requêtes*, qui sont donc toutes soumises à la procédure de *chase*, très coûteuse, pour vérifier leur équivalence avec la requête à reformuler.

Malheureusement, en plus d'être coûteuse, cette procédure est souvent appliquée pour uniquement constater par la suite la non-existence d'un *containment mapping*, et par conséquent rejeter la sous-requête en question. On note aussi une importante redondance dans la *chase* des sous-requêtes qui ont des parties communes, redondance que par sa construction le $C\&B$ ne peut pas éviter.

Dans cette thèse, on présente une approche alternative de la phase coûteuse et pénalisante de backchase. Notre approche consiste essentiellement à effectuer pendant la phase de backchase *une seule séquence de chase* sur le plan universel. On va ainsi *éviter le nombre exponentiel de séquences de chase sur les sous-requêtes*, qui constitue le problème de base dans le manque de performances du $C\&B$. On va aussi, implicitement, éviter les séquences de *chase* inutiles et la redondance.

Alors que cette idée semble très prometteuse pour gagner en performance, la question naturelle qui se pose est celle de savoir comment cette unique séquence de *chase* peut-elle nous permettre de retrouver les reformulations minimales. En effet, la séquence de *chase* effectuée par le $C\&B$ sur le plan universel permet juste de décider si le plan universel est lui-même une reformulation.

La réponse à cette question est que notre séquence de *chase* sera en revanche *enrichie* avec des *annotations de provenance*. Le but de ces annotations sera de *montrer directement*, pour chaque atome (élément d'une requête) comment cet atome peut être obtenu en appliquant des procédures de *chase* à des sous-requêtes du plan universel.

Ainsi, le point de départ de notre séquence de *chase* annotée sera un plan universel où chaque élément de la clause FROM (atome) est annoté avec *lui-même*, au moyen de sa variable (qui l'identifie de façon unique). Ce type d'annotations combinées vont servir à déterminer des sous-requêtes : en effet, une sous-requête est identifiée de façon unique par un sous-ensemble des

éléments de la clause FROM du plan universel, donc par un ensemble de ces annotations initiales.

Le comportement qu'on souhaite est celui de pouvoir identifier, à la fin de la séquence de *chase* annotée, par les *containment mappings* depuis la requête initiale vers le résultat annoté de la séquence, toutes les sous-requêtes qui sont des reformulations minimales. Ceci sera effectué en combinant les annotations individuelles des atomes dans l'image d'un *containment mapping*. Pour mieux comprendre le procédé global qu'on souhaite, on montre ci-dessous l'intuition derrière notre approche, en reprenant notre exemple de l'editeur de logiciel.

Exemple B.2.4. *Continuant avec notre exemple de problème de réécriture, la séquence de chase annotée commencera avec le plan universel où les atomes correspondant aux vues sont annotés avec eux mêmes (la variable qui leur correspond de façon unique). Par la suite, les atomes correspondant aux relations R, S et T sont annotés de façon à identifier l'atome initial qui, par un chase step avec la contrainte correspondante, a été responsable de leur ajout dans la requête :*

$U^{\mathcal{C}}$ *: select* v_r.A
from $V_R\ v_r[v_r]$, $V_S\ v_s[v_s]$, $V_T\ v_t[v_t]$, $V_{RS}\ v_{rs}[v_{rs}]$,
$R\ r_1[v_{rs}]$, $S\ s_1[v_{rs}]$, $R\ r_2[v_r]$, $S\ s_2[v_s]$, $T\ t[v_t]$
where v_r.C=v_s.C *and* v_s.D=v_t.D *and* v_r.A=v_{rs}.A *and* v_s.D=v_{rs}.D *and* r_1.A=v_{rs}.A
and s_1.D=v_{rs}.D *and* r_1.C=s_1.C *and* r_2.A=v_r.A *and* r_2.C=v_r.C
and s_2.C=v_s.C *and* s_2.D=v_s.D *and* t.D = v_t.D *and* t.E = v_t.E

On rappelle qu'il existe deux containment mappings de Q à $U^{\mathcal{C}}$. Le premier est h_1, comprenant dans son image r_1, s_1 et t, et le deuxième est h_2, avec r_2, s_2 et t. Ces mappings et les annotations de leurs images fournissent donc les reformulations minimales comme suit : le premier mapping fournit v_{rs} (deux fois) et v_t, ce qui donne R_2. Le deuxième mapping fournit R_1 par v_r, v_s et v_t.

L'exemple ci-dessus donne une idée globale de notre approche, montrant comment après une seule séquence de *chase*, propageant les annotations conformément aux *chase steps*, on arrive à *directement lire* les reformulations recherchées. Dans ce cas très simple, il suffit en effet d'annoter la procédure standard de *chase* pour obtenir le bon résultat.

Malheureusement dans le cas général, ceci n'est pas possible, car la procédure standard de *chase* est *trop agressive* dans son application. En effet, dans la procédure standard de *chase*, un *chase step* ne s'applique pas si les atomes correspondants sont considérés comme déjà existants. La notion d'identité sur les atomes (cette notion qui identifie un atome à ajouter comme étant identique à un atome déjà présent, et donc considère l'atome à ajouter comme *déjà existant*) est dans le cadre de la *chase* standard *insuffisamment granulaire pour nos besoins d'annotation*, comme le montre l'exemple suivant :

Exemple B.2.5. *Considérons le schéma R(A), S(B, C, D), et la requête et les vues suivantes :*

Q : *select* $r.A$ *from* $R\ r, S\ s$ *where* $s.B = r.A$ *and* $s.C = 1$ *and* $s.D = 2$
$V_1(A)$: *select* $r.A$ *from* $R\ r, S\ s$ *where* $s.B = r.A$ *and* $s.C = 1$ *and* $s.D = 2$
$V_2(A)$: *select* $r.A$ *from* $R\ r, S\ s$ *where* $s.B = r.A$ *and* $s.C = 1$

Pour le problème de réécriture totale avec des vues, le plan universel initialement annoté sera :

U : *select* $v_1.B$ *from* $V_1\ v_1[v_1], V_2\ v_2[v_2]$ *where* $v_1.A = v_2.A$

En instrumentant directement la chase standard avec provenance on obtient d'abord, en effectuant un chase step avec la contrainte associée à V_1 *:*

$\forall\ v_1 \in V_1 \longrightarrow \exists\ r \in R, s \in S,\ r.A = v_1.A$ *and* $s.B = r.A$ *and* $s.C = 1$ *and* $s.D = 2$

le résultat :

U' : *select* $v_1.A$ *from* $V_1\ v_1[v_1], V_2\ v_2[v_2], R\ r_1[v_1], S\ s_1[v_1]$
where $v_1.A = v_2.A$ *and* $r_1.A = v_1.A$ *and* $s_1.B = r.A$ *and* $s_1.C = 1$ *and* $s_!.D = 2$

. *En essayant d'appliquer ensuite un chase step avec la contrainte associée à* V_2 *:*

$\forall\ v_2 \in V_2 \longrightarrow \exists\ r \in R, s \in S,\ r.A = v_2.A$ *and* $s.B = r.A$ *and* $s.C = 1$

ce chase step ne sera pas appliqué, car pour la chase standard les atomes correspondants sont déjà présents. *Dans ce cas la seule solution de notre stratégie d'annotation est celle d'*enrichir *l'annotation des atomes déjà présents, comme suit :*

U'' : *select* $v_1.A$ *from* $V_1\ v_1[v_1], V_2\ v_2[v_2], R\ r_1[v_1 + v_2], S\ s_1[v_1 + v_2]$
where $v_1.A = v_2.B$ *and* $r_1.A = v_1.A$ *and* $s.B = r.A$ *and* $s.C = 1$ *and* $s.D = 2$

En essayant de lire les reformulations en utilisant le containment mapping de Q *vers* U'' *on va donc obtenir comme reformulations possibles la sous-requête correspondant à* v_1 *et celle correspondant à* v_2*. Mais cela est faux, car la sous-requête correspondant à* v_2 *n'est pas équivalente à Q ! En effet, dans la vue* V_2 *il manque la sélection avec* 2 *sur* $s.D$*. Les annotations de provenance propagées en annotant directement la chase standard sont donc incorrectes.*

La Conservative Chase. Pour obtenir des annotations correctes, il faudrait donc trouver un autre type de procédure de *chase*, qui soit capable de mieux *renforcer l'identité des atomes*. Dans ce but, nous proposons dans cette thèse la Conservative Chase. Cette nouvelle procédure de *chase* est basée sur une modification des contraintes pour *renforcer l'identité des attributs non-déterminés* (c'est-à-dire les attributs des atomes de la conclusion qui ne sont pas rendus égaux dans la contrainte à un attribut de la prémisse ou à une constante). En effet, on peut montrer que c'est précisément la présence dans les contraintes de tels attributs qui empêche la *chase* standard de fonctionner correctement avec des annotations de provenance.

Ce renforcement d'identité correspond à un procédé classique en logique du premier ordre, la Skolemisation. Les attributs non déterminés sont rendus égaux à des *termes Skolem*, qui consistent en un symbole de fonction et des arguments. Les symboles de fonctions doivent etre distincts pour des contraintes distinctes. Le choix d'arguments dans notre cas correspond aux attributs dans la prémisse qui apparaissent aussi dans les égalités de la conclusion. Ce choix d'arguments peux sembler surprenant, néanmoins il fournit la base de nos résultats théoriques exposés par la suite.

Nous présentons ci-dessous la forme *Skolemisée* (appelée par la suite *sk_form*) des contraintes qui s'appliquent dans la *chase* annotée du plan universel, dans l'exemple de l'éditeur de logiciel :

$$
\begin{array}{ll}
b_{V_R} : \forall v_r, v_r \in V_R & \rightarrow \exists r, r \in R \wedge r.A = v_r.A \wedge r.C = v_r.C \\
 & \quad \wedge \mathbf{r.B = f_1(v_r.A, v_r.C)} \\
b_{V_S} : \forall v_s, v_s \in V_S & \rightarrow \exists s, s \in S \wedge s.C = v_s.C \wedge s.D = v_s.D \\
b_{V_{RS}} : \forall v_{rs}, v_{rs} \in V_{RS} & \rightarrow \exists r, s,\ r \in R \wedge s \in S \wedge r.A = v_{rs}.A \wedge s.D = v_{rs}.D \\
 & \quad \wedge \mathbf{r.C = f_2(v_{rs}.A, v_{rs}.D)} \\
 & \quad \wedge \mathbf{s.C = f_2(v_{rs}.A, v_{rs}.D)} \\
 & \quad \wedge \mathbf{r.B = f_3(v_{rs}.A, v_{rs}.D)} \\
b_{V_T} : \forall v_t, v_t \in V_T & \rightarrow \exists t, t \in T \wedge t.D = v_t.D \wedge t.E = v_t.E
\end{array}
$$

En partant de la *sk_form* d'un ensemble de contraintes, nous obtenons par la suite leur *sk_unit_form*, qui est un ensemble de contraintes à un seul atome principal dans leur conclusion, et qui est le résultat d'une *division* des contraintes en *sk_form* en plusieurs contraintes *unitaires*.

La Conservative Chase avec la *sk_form* ou la *sk_unit_form* d'un ensemble de contraintes demande à ce que deux atomes soient considérés identiques uniquement si tous leur attributs sont égaux. Alors que la *chase* standard *ignore cette propriété pour les attributs non-déterminés*, la Conservative Chase utilise les termes Skolem correspondants et *demande à ce que les termes Skolem soient égaux aussi*. Malgré cette apparente divergence de comportement entre les deux procédures, nous prouvons dans cette thèse le résultat suivant, qui pose la base de correction de notre nouvel algorithme de reformulations :

Théorème B.2.6. *Le résultat d'une séquence complète de* chase *standard et celui d'une séquence complète de Conservative Chase (avec la* sk_form *ou la* sk_unit_form *des contraintes originales) sont équivalents.*

D'un autre coté, le choix des arguments des termes Skolem est essentiel pour assurer une autre propriété très importante de la Conservative Chase : sa terminaison pour des contraintes *faiblement acycliques*. En effet, il est bien connu que pour un ensemble arbitraire de contraintes, les séquences de *chase* standard peuvent être infinies. Ceci est également vrai pour la Conservative Chase. Une des conditions les moins restrictives et les plus utilisées sur les contraintes, qui assure que toutes les séquences complètes de *chase* standard soient finies, est la *faible acyclicité* de l'ensemble de contraintes. Par sa construction, la Conservative Chase garantit la même propriété, comme suit :

Théorème B.2.7. *Pour un ensemble faiblement acyclique de contraintes, toute séquence complète de Conservative Chase (avec leur* sk_form *ou leur* sk_unit_form*) est finie, et comporte la même borne supérieure que la* chase *standard, polynomiale dans la taille de la requête initiale.*

La Provenance-Aware Chase (pa_chase**).** En reprenant notre idée initiale d'annotations, mais en utilisant à la place de la *chase* standard *la Conservative Chase avec la sk_unit_form des contraintes*, on obtient la forme finale de notre procédure de *chase* annotée, appelée la Provenance-Aware Chase. C'est donc la pa_chase qu'on appliquera en une seule séquence au plan universel dans notre algorithme de reformulations.

Dans la pa_chase, les annotations de provenance sont maintenues en tant que formules booléennes en FND (forme normale disjonctive) : chacune des conjonctions dans la formule

associée à un atome dénotera une partie du plan universel (sous-requête) qui par sa *chase* standard individuelle peut générer l'atome en question. Ces conjonctions peuvent être combinées par l'opération logique de *et* pour construire d'autres conjonctions et ainsi designer d'autres sous-requêtes.

L'application d'un *chase step* dans le cas de la Provenance-Aware Chase transfère (copie) la provenance de l'image de la prémisse (qui est la conjonction des atomes dans l'image) sur la conclusion. Si la conclusion n'est pas présente, elle sera ajoutée. Dans le cas contraire, son annotation sera *enrichie* avec la nouvelle formule, au moyen d'un *ou* logique, et la formule résultante sera une disjonction. On rappelle que la notion de conclusion *déjà présente* est celle de la Conservative Chase, et emploie les termes Skolem.

Lecture des reformulations. Une fois la Provenance-Aware Chase du plan universel terminée, on lit les reformulations minimales comme suggéré dans les exemples précédents, mais avec une étape supplémentaire, comme suit :

- on calcule tous les *containment mappings* de Q au résultat de la pa_chase du plan universel
- on calcule la FND de la disjonction des formules des images de ces *mappings*. On rappelle que chacune des formules d'une image est elle-même la conjonction logique des formules individuelles des atomes dans l'image.
- on calcule la *forme réduite* de cette FND, en y enlevant toutes les conjonctions qui *impliquent logiquement d'autres conjonctions déjà présentes dans la FND*.

A noter, la dernière étape ci-dessus n'est pas visible sur notre exemple, mais elle est nécessaire dans le cas général, car la formule FND construite peut contenir des réécritures *valides mais non minimales*. Alors que la vérification de la minimalité d'une reformulation peut en général être très coûteuse, nos opérations s'effectuent uniquement sur des formules booléennes et l'élimination des conjonctions non-minimales est ainsi très rapide.

L'algorithme de reformulation Provenance-Aware Chase & Backchase. En se basant sur les concepts présentés précédemment, on montre ci-dessous une vue haut-niveau de notre nouvel algorithme de reformulation $Prov_{C\&B}$:

$Prov_{C\&B}$ (schéma source S, schéma cible T,
ensemble faiblement acyclique de contraintes $\mathcal{C}$, requête à reformuler Q)

//étape de chase :
1. calcul du plan universel U comme dans le $C\&B$
   par une séquence de chase standard sur Q avec $\mathcal{C}$ et retenant ensuite uniquement les atomes de T

//recherche des reformulations (remplace l'étape de backchase) :
2. calcul du résultat U' d'une séquence de pa_chase sur U avec la *sk_unit_form* de $\mathcal{C}$
3. calcul de l'ensemble $\mathcal{H}$ de tous les *containment mappings* de Q à U'

4. calcul de Π, la FND de $\bigvee_{h\in\mathcal{H}} \pi(h(Q))$ ($\pi(h(Q))$ représente la provenance de l'image)
5. calcul de la forme réduite $rf(\Pi)$ de Π
6. résultat : toutes les sous-requêtes de U correspondant aux conjonctions présentes dans $rf(\Pi)$.

Le principal résultat théorique concernant notre algorithme de reformulation est le suivant :

Théorème B.2.8. *$Prov_{C\&B}$ est correct et complet : il trouve* toutes *et* exactement *les reformulations minimales de la requête initiale sous les contraintes données.*

Comme souligné précédemment, la correction et la complétude étaient dans notre démarche des buts obligatoires dans la conception de notre nouvelle technique de recherche de reformulations minimales. En revanche, un but essentiel était également celui d'atteindre des *performances significatives sur le plan pratique*.

Évaluation expérimentale. Pour estimer ces performances, on a procédé à une implémentation de notre algorithme, dont on a mesuré le comportement pratique. Pour ce faire, on a re-créé et étendu le cadre expérimental présenté dans [60]. On a choisi ce cadre pour son intérêt pratique, sa facilite de paramétrisation et aussi pour sa capacité à fournir un outil de *stress-test* pour notre algorithme.

Notre contexte expérimental consiste donc en des schémas et requêtes de type *chaîne-d'étoiles* [60], dont la forme la plus simple est la suivante :

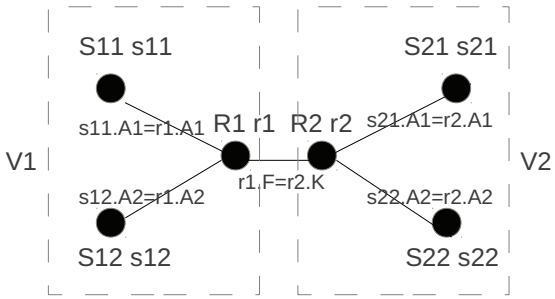


FIGURE B.1: Schéma et requête de type chaîne-d'étoiles avec deux étoiles à deux coins.

Ce type de schémas et de requêtes peut être *paramétré par le nombre d'étoiles et le nombre de leur coins*. Il comporte des contraintes provenant des vues, qui couvrent chacune *une combinaison centre + deux coins consécutifs d'une étoile*, ainsi que des contraintes de clé (sur l'attribut K qui est spécifique au centre d'une étoile) et de clé étrangère entre deux centres consécutifs dans la chaîne. Le problème posé est celui de trouver toutes les réécritures minimales *partielles* avec les vues et sous les contraintes. Il s'agit ainsi de réécritures qui peuvent utiliser aussi bien les vues que les tables originales (centres et coins des étoiles).

Notre première mesure vise à déterminer **l'intérêt global sur le plan pratique de notre algorithme**. Dans ce contexte, on a effectué une comparaison avec un SGBD commercial, en calculant le rapport entre :

- le temps que le SGBD prend pour trouver une reformulation et l'exécuter, quand on lui fournit directement la requête initiale.
- le temps nécessaire pour que notre algorithme trouve toutes les reformulations minimales + le temps (en réalité négligeable) de choisir parmi elles une de celles qui comporte le moindre nombre d'éléments dans la clause FROM (stratégie heuristique visant a choisir une reformulation de coût très bas, potentiellement optimal) + le temps d'exécuter cette reformulation dans le SGBD.

On présente ci-dessous la moyenne des mesures obtenues pour ce rapport (appelée *avg speedup factor*) sur 10 instances de base de données générées aléatoirement. La population des bases de données assure les propriétés suivantes : l'évaluation des reformulations est de façon significative plus rapide que celle des requêtes, et plus une reformulation emploie des vues, plus son coût décroit, ce qui correspond aux scénarios pratiques d'utilisation des vues matérialisées dans l'optimisation des requêtes. Comme spécifié précédemment, les requêtes (de type chaîne-d'étoiles) sont paramétrées par leur nombre d'étoiles et de coins :

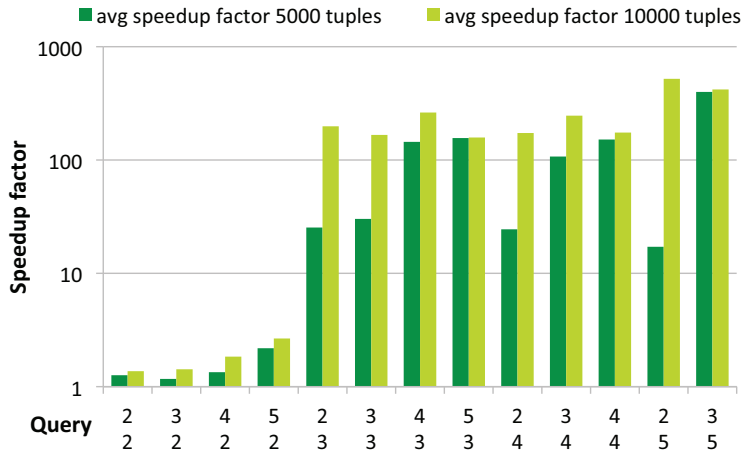


FIGURE B.2: Facteurs de gain moyens sur 10 instances de bases de données

On note l'écart impressionnant entre les performances en utilisant notre algorithme et celles obtenues en se basant uniquement sur le SGBD. Le gain de performances induit par notre algorithme va **jusqu'à deux ordres de magnitude**, et manifeste une tendance croissante avec la taille de la base de données. Ce gain est essentiellement dû à la *complétude* de notre algorithme, qui permet de proposer une reformulation de coût très bas, par rapport à la meilleure reformulation trouvée par le SGBD, dont l'algorithme est *incomplet*, et qui trouve donc une reformulation restant coûteuse à s'exécuter (cette reformulation n'emploie pas toutes les vues disponibles).

D'un autre coté, la même reformulation peu coûteuse trouvée par le $Prov_{C\&B}$ en raison de sa complétude, aurait pu être trouvée en employant le $C\&B$ original, car on rappelle que le $C\&B$ est lui-aussi complet. Le gain aurait en revanche été perdu car le $C\&B$ aurait mis un temps de calcul beaucoup trop long pour justifier l'intérêt du gain en exécution. En effet, l'intérêt de notre algorithme est celui de *préserver la complétude à un coût qui ne la rende pas inutile !*

Notre deuxième mesure vise justement **le temps passé par notre algorithme pour trouver toutes les reformulations minimales**. On reprend ainsi le cadre expérimental précèdent, et on l'enrichit aussi avec des contraintes de clé étrangère et des tables supplémentaires, pour obtenir ce qu'on appelle *une configuration chaîne d'étoiles étendue* :

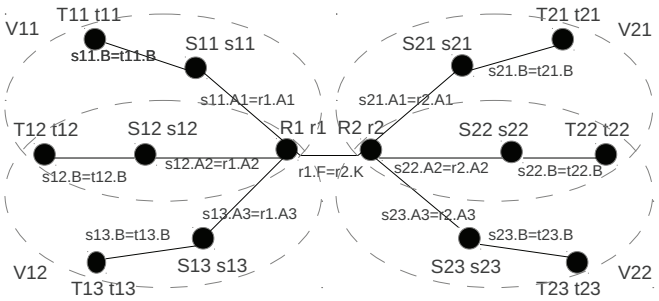


FIGURE B.3: Chaîne d'étoiles étendue

Pour les deux types de configurations et pour des requêtes paramétrées de la même façon par le nombre d'étoiles et de coins, on présente ci-dessous le temps nécessaire à $Prov_{C\&B}$ pour trouver toutes les reformulations minimales :

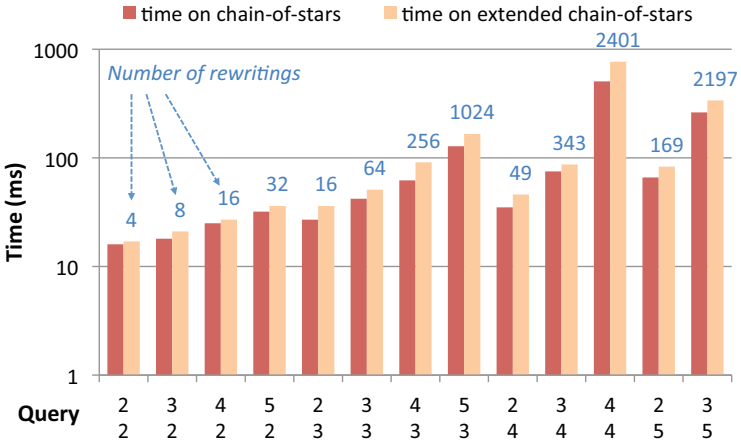


FIGURE B.4: Temps de calcul de toutes les reformulations minimales

.

Ces résultats montrent que notre algorithme est non seulement utile mais aussi *très rapide*, trouvant en moins d'une seconde toutes les reformulations minimales, *même dans les cas ou il en existe des milliers* (ces cas font partie de notre évaluation dans le but de *stress-test*). Sa rapidité est d'autant plus soulignée en la comparant aux temps d'exécution des requêtes sur le SGBD, qui dans les cas analysés sont de l'ordre des minutes.

Rafinement de $Prov_{C\&B}$ pour les reformulations de coût minimum. Dans la plupart des cas en pratique, l'intérêt de trouver les reformulations minimales est, comme souligné précédemment, celui de retrouver parmi elles celles de *coût minimum*. Dans la version de base de notre

algorithme, ceci peut être réalisé en trouvant d'abord toutes les reformulations minimales, puis en mesurant leur coût et en choisissant celle du moindre coût. C'est en effet la stratégie appliquée dans notre évaluation expérimentale précédente, où la fonction de coût correspond au nombre d'éléments dans la clause FROM.

En revanche, par sa construction, notre algorithme permet aussi de trouver de façon *plus directe* les reformulations de cout minimum *dans le cas des fonctions de coût monotones*. Pour cela, on procède à une *adaptation de la Provenance-Aware Chase* basée sur les idées suivantes :

- une conjonction correspondant à une reformulation de coût non-minimum *n'est pas utile à maintenir* dans les formules de provenance
- le calcul de reformulations peut être *entrelacé* avec la Provenance-Aware Chase, de façon à raffiner une quantité *seuil*, représentant intuitivement le coût de la meilleure reformulation trouvée jusqu'au $pa\_chase\ step$ courant. Ce coût sera bien sur supérieur ou égal au coût minimum et donc peut servir pour filtrer les conjonctions comme indiqué précédemment.

Ces observations donnent lieu à une version modifiée et allégée de notre algorithme, qu'on appelle *PRUNED* $Prov_{C\&B}$ et dont les performances sont présentées ci-dessous, pour la même fonction de coût que précédemment (le nombre d'éléments de la clause FROM), et les mêmes requêtes de type chaîne-d'étoiles :

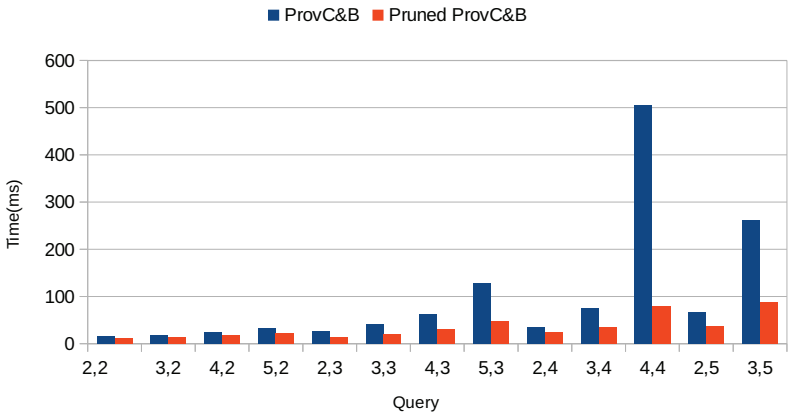


FIGURE B.5: Comparaison de $Prov_{C\&B}$ et PRUNED $Prov_{C\&B}$

On note que la version adaptée de notre algorithme permet d'obtenir des temps d'exécution jusqu'à six fois plus bas que sa version initiale (qui exhibait déjà, on le rappelle, des performances significatives).

L'ensemble de nos mesures nous permet d'affirmer que, malgré les opinions précédentes, *la complétude (avec ses effets extrêmement bénéfiques sur les performances) est atteignable en pratique, qui plus est à un coût très bas, et largement contrebalancé par les gains obtenus*. On estime aussi que les gains (déjà impressionnants) en performances peuvent être encore plus accentués par une intégration de nos techniques directement dans les SGBD, évitant ainsi les coûts d'interface entre la recherche de la reformulation et son exécution.

B.3 Condensé du deuxième chapitre

Dans le deuxième chapitre de cette thèse nous revisitons, développons et améliorons le travail de Cautis, Deutsch et Onose, présenté dans [16] et détaillé dans [56], sur le problème des réécritures de requêtes XPath comprenant un seul niveau d'intersection de plusieurs vues.

B.3.1 Réécritures à un seul niveau d'intersection

Nous commençons par rappeler le problème de la réécriture à un seul niveau d'intersection de plusieurs vues [16], et les concepts qui permettent de définir ce problème.

Documents XML. Un document XML D est considéré comme un arbre dont les nœuds sont étiquetés avec des symboles d'un alphabet infini Σ, et dont la racine (le *nœud document*) est étiquetée avec *doc(D)* (on considère que tous les symboles de type *doc(nom)* font partie de Σ).

Requêtes et vues, tree patterns. Le sous-langage de XPath considéré pour les requêtes et les vues dans [16] et [56] est appelle *XP* et correspond à des requêtes XPath sans *wildcards*, avec de la navigation enfant (/) et descendant (//), en partant de la racine d'un document donné D (commençant donc par *doc(D)*). Les vues sont des requêtes dont le résultat a été matérialisé dans un document qui porte leur nom.

Les requêtes dans *XP* sont représentables par des *tree patterns*. Un *tree pattern* p [16] est un arbre comprenant un ensemble de nœuds $\text{NODES}(p)$ étiquetés avec des symboles de Σ par une fonction λ_p, un nœud special racine $\text{ROOT}(p)$ et un noeud special output $\text{OUT}(p)$, et des arêtes enfant (/) ou descendant (//).

Intersections et DAG patterns. Pour formaliser les intersections, nous considérons deux langages : $XP^{\cap-simple}$ qui comprend de la navigation suivie par une intersection, et $XP^{\cap}$ [16] qui comprend de la navigation, une intersection, et potentiellement de la navigation supplémentaire. Le langage $XP^{\cap-simple}$ ne figure pas dans les analyses précédentes, mais il est nécessaire dans le travail que nous présentons ici pour structurer les résultats théoriques.

Sous certaines conditions (même étiquette pour la racine et l'output) les requêtes avec intersection sont représentables par des *DAG patterns*. Un *DAG pattern* d [16] est un graphe orienté acyclique, comprenant un ensemble de nœuds $\text{NODES}(d)$ étiquetés avec des symboles de Σ par une fonction λ_d, un nœud special racine $\text{ROOT}(p)$ et un nœud special output $\text{OUT}(d)$, et des arêtes de type enfant (/) ou descendant (//). Un DAG pattern a les propriétés suivantes : chacun de ses nœuds est accessible depuis la racine, et les nœuds depuis lesquels $\text{OUT}(d)$ n'est pas accessible, appelés des nœuds *prédicat*, ont une seule arête rentrante. On utilisera par la suite le terme *pattern* pour designer les *tree patterns* et les *DAG patterns*.

Branches principales. Dans un *pattern*, un chemin entre le nœud racine et le nœud output est appelé une branche principale, et les nœuds sur ce type de chemins sont appelés nœuds de branche principale. Pour un chemin p_1 comprenant une partie de branche principale dans un pattern d, on dénote par $\text{TP}_d(p_1)$ le *tree pattern* qui a comme branche principale p_1.

Fragments intéressants de XP. [16] distingue deux fragments intéressants de *XP*, pour lesquels on peut obtenir des résultats théoriques supplémentaires. Le premier de ces fragments, appelé XP_{es}, dont nous présentons ici un raffinement, correspond aux requêtes de type *extended skeletons*. Ces requêtes sont telles que dans le *tree pattern* correspondant pour chaque sous-prédicat de type // rattaché à un nœud de branche principale différent de l'output, il n'existe aucun *mapping* entre la branche / rentrante du prédicat et la branche / sortante du nœud.

Le fragment $XP_{//}$ étend le fragment XP_{es} ; les //-prédicats directement rattachés aux nœuds de branche principale y sont autorisés et l'usage des //-sous-prédicats à l'intérieur de ces prédicats est libre.

Réécritures dans $XP^{\cap}$. [16] et [56] se focalisent sur le probleme des reecritures dans $XP^{\cap}$. Pour un ensemble $D_{\mathcal{V}}$ de documents correspondant à des vues définies sur un document D, un *plan de réécriture* dans $XP^{\cap}$ est une requête dans $XP^{\cap}$ utilisant les documents des vues. L'*expansion* d'un tel plan r, notée *unfold*(r), est une requête dans $XP^{\cap}$ ou chaque $doc(v_i)$ est remplacé par la définition de la vue v_i. Un plan de réécriture est appelé *une réécriture* d'une requête q si son expansion est équivalente à q (ils produisent les mêmes résultats sur tous les documents).

Un algorithme correct et complet de réécriture. [16] présente un algorithme de réécriture où la navigation est réalisée par la fonction *compensate* (qui concatène de la navigation sous la forme d'une partie de requête existante) et un *préfixe sans perte* est un préfixe de la branche principale d'un *tree pattern* tel que le suffixe restant est transformé en prédicat. On présente ci-dessous la forme clarifiée de l'algorithme REWRITE qui vise à trouver une réécriture dans $XP^{\cap}$ pour une requête q et un ensemble de vues $\mathcal{V}$:

REWRITE$(q, \mathcal{V})$
1 **for** p un préfixe sans perte de *pattern*(q)
2 **do**
3 $r \leftarrow$ BUILDINITREWRITECANDIDATE$(p, \mathcal{V})$
4 $d \leftarrow$ *pattern*(*unfold*(r))
5 **if** $d \equiv p$
6 **then return** compensate$(r, q,$ OUT$(p))$
7 **return** $\emptyset$

BUILDINITREWRITECANDIDATE$(p, \mathcal{V})$
1 $\mathcal{V}' \leftarrow \phi$
2 **for** $v \in \mathcal{V}$, h un *mapping racine* de *pattern*$(\bar{v})$ a p
3 **do**
4 $b \leftarrow h($OUT(*pattern*$(\bar{v})))$
5 $\mathcal{V}' \leftarrow \mathcal{V}' \cup$ compensate (doc("v")/$\lambda_p(b), p, b)$
6 $r \leftarrow \left(\bigcap_{v_j \in \mathcal{V}'} v_j\right)$
7 **return** r

B.3.2 Réécritures, équivalence et union-freeness

Malheureusement, malgré sa correction et complétude montrées dans [16] et [56], REWRITE n'est pas performant. En effet, il est affirmé dans [16] et prouvé dans [56] que *le problème de réécriture en XP$^{\cap}$ est coNP-complet*. On va par la suite montrer comment cette analyse du problème de réécriture nous permet d'investiguer deux autres problèmes : *l'équivalence DAG-tree* et la *union-freeness d'un DAG*.

Equivalence DAG-tree. Une brique principale de REWRITE consiste dans le test d'équivalence entre un *DAG pattern* dans *XP*$^{\cap-simple}$ et un *tree pattern*. Ce test est effectué un nombre de fois qui correspond aux nombre de préfixes de la requête initiale. En se basant sur cette remarque, on peut donc affirmer que le problème de la réécriture a une *réduction polynomiale* au problème de décision de l'équivalence entre une requête exprimée par un *DAG pattern* dans *XP*$^{\cap-simple}$ et une requête exprimée par un *tree pattern*. On va appeler par la suite ce problème *le problème de l'équivalence DAG-tree*.

Union-freeness. On montre également que le problème de l'équivalence DAG-tree a une réduction polynomiale au *problème de la union-freeness d'un DAG*, qui consiste à trouver un *tree pattern* équivalent à un *DAG pattern* donné, si un tel *tree pattern* existe. On peut ainsi *étendre les résultats de complexité du problème de réécriture* pour caractériser les problèmes d'équivalence et union-freeness comme suit :

Théorème B.3.1. *Le problème de l'équivalence entre un DAG pattern dans XP*$^{\cap-simple}$ *et un tree pattern est coNP-complet. Le problème de la union-freeness d'un DAG pattern dans XP*$^{\cap-simple}$ *est coNP-difficile.*

Les liens entre les problèmes montrés ci-dessus nous permettent aussi de *baser la résolution du problème de réécriture sur la résolution de la union-freeness*. Pour trouver une solution au problème de la *union-freeness*, on va utiliser la notion de *interleaving*, qui consiste intuitivement en un *pliage* d'un *DAG pattern* en un arbre (*tree pattern*).

En effet, il a été montré qu'un *DAG pattern* est équivalent à l'union de ses *interleavings*. Il s'en suit qu'un *DAG pattern* possède un *tree pattern équivalent* si et seulement si il possède un *interleaving dominant*, c'est-à-dire un *interleaving qui contient tous les autres*. Cet *interleaving dominant* constitue donc une solution au problème de la *union-freeness*. Une façon naïve de résoudre le problème de la *union-freeness* est donc en employant l'algorithme suivant :

DOMINANT_INTERLEAVING(d)

1 génération de tous les *interleavings* de d
2 vérification de l'existence d'un *interleaving* qui contient tous les autres
3 si c'est le cas, retourner l'*interleaving* dominant, sinon retourner $\emptyset$

On appelle l'approche ci-dessus naïve car pour un *DAG pattern* donné il peut y avoir un *nombre exponentiel* de *interleavings*, correspondant à un nombre exponentiel de *pliages* possibles. La génération de tous ces *interleavings* et leur comparaison peuvent donc être très coûteuses.

B.3.3 Un algorithme à base de règles pour construire le *interleaving* dominant

Comme alternative moins coûteuse à la procédure naïve qui consiste à générer et tester tous les *interleavings*, [16] propose un algorithme qui consiste en l'application itérative d'un *ensemble de règles*, visant à transformer le *DAG pattern* progressivement, jusqu'à en obtenir le *interleaving dominant*. Chacune de ces règles effectue ainsi intuitivement un *pliage* du *DAG pattern* courant.

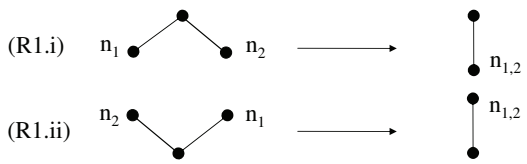
Dans cette thèse nous reprenons et améliorons cet algorithme, pour assurer sa *complexité polynomiale* et améliorer les conditions de complétude résultantes. Nous présentons ci-dessous la forme globale de l'algorithme modifié, tout en gardant son nom précèdent, APPLY-RULES.

APPLY-RULES(d)

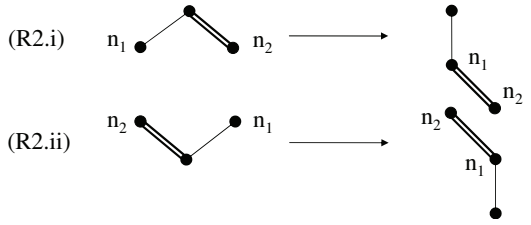
```
1  d' = d
2  if un des tree patterns intersectés dans d est un /-pattern
3     then RuleSet = R1, R2, R3, R4, R6, R7
4     else RuleSet = R1, R2, R3, R4, R5, R6
5  repeat
6           while R1 s'applique sur d'
7                 do
8                    d' = appliquer R1 sur d'
9           if une règle Ri dans RuleSet s'applique sur d'
10             then d' = appliquer Ri sur d'
11             else break
12 return d'
```

Dans notre version de APPLY-RULES, nous avons modifié le flux global et raffiné les cas d'utilisation des règles. Nous avons également remplacé deux des règles précédentes par une nouvelle règle, qui les englobe et qui possède la propriété essentielle d'être testable en temps polynomial. Nous présentons ci-dessous l'ensemble résultant des règles :

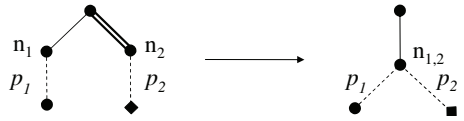
Règle R1. Cette règle s'applique quand $\lambda_d(n_1) = \lambda_d(n_2)$.



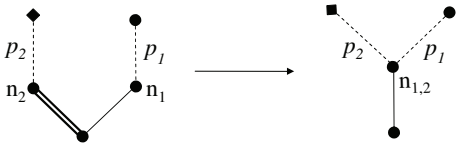
Règle R2. Cette règle s'applique si n_1 et n_2 ne sont pas unifiables et n_2 n'est pas atteignable depuis n_1 (resp. n_1 n'est pas atteignable depuis n_2, dans le cas de R2.ii).



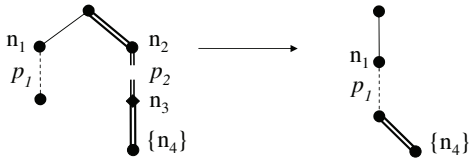
Règle R3.i. Cette regle s'applique sous les conditions suivantes : (1) $p_1 \equiv p_2$, (2) chacun des noeuds de p_2 a une seule arête de branche principale rentrante, (3) $\text{TP}_d(p_2)$ contient $\text{TP}_d(p_1)$.



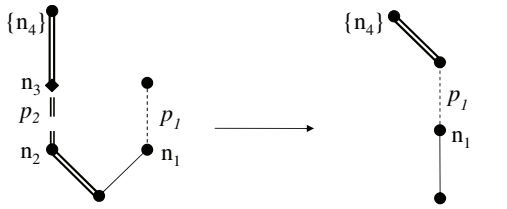
Règle R3.ii. Cette règle s'applique sous les conditions suivantes : (1) $p_1 \equiv p_2$, (2) chacun des nœuds de p_2 a une seule arête de branche principale sortante, (3) $\text{TP}_d(p_2)$ contient $\text{TP}_d(p_1)$.



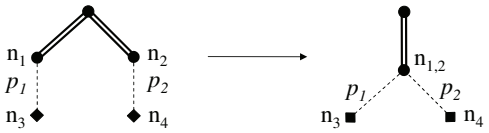
Règle R4.i. Cette règle s'applique sous les conditions suivantes, qui doivent être valides *pour tous les nœuds* n_4 : (1) n_3 a une seule arête rentrante de branche principale, les autres nœuds de p_2 ont une seule arête de branche principale rentrante et une seule arête de branche principale sortante, (2) il existe un *mapping* de $\text{TP}_d(p_2)$ vers $\text{SP}_d(n_1)$, tel que l'image de tous les nœuds de p_2 est dans p_1, (3) le chemin $p_2//n_4$ n'a pas de *mapping* dans p_1.



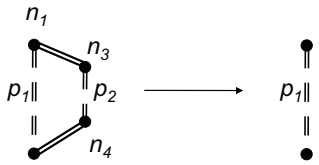
Règle R4.ii. Cette règle s'applique sous les conditions suivantes, qui doivent être valides *pour tous les nœuds* n_4 : (1) n_3 a une seule arête de branche principale sortante, tous les autres nœuds de p_2 ont une seule arête de branche principale rentrante et une seule arête de branche principale sortante, (2) il existe un *mapping* de $\text{TP}_d(p_2)$ vers $\text{TP}_d(p_1)$, tel que l'image de tous les nœuds de p_2 est dans p_1, (3) le chemin $n_4//p_2$ n'a pas de *mapping* dans p_1.



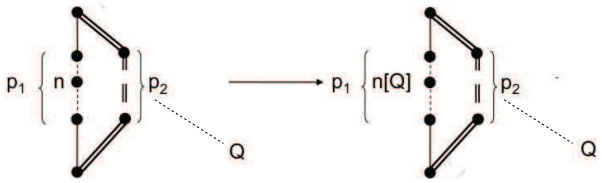
Règle R5. Cette règle (anciennement R6) s'applique sous les conditions suivantes : (1) n_3, n_4 ont une seule arête de branche principale rentrante, tous les autres nœuds de p_1 et p_2 ont une seule telle arête rentrante et une seule sortante, (2) $\text{TP}_d(p_1)$ et $\text{TP}_d(p_2)$ sont *similaires* [16].



Règle R6. Cette règle (anciennement R7) s'applique sous les conditions suivantes : (1) les nœuds de p_2 ont une seule arête de branche principale rentrante et une seule arête de branche principale sortante, (2) il existe un *mapping* de $\text{TP}_d(p_2)$ vers $\text{SP}_d(n_1)$, tel que l'image des nœuds de p_2 est dans p_1.



Règle R7. Cette règle est nouvelle. Elle remplace, englobe et corrige les précédentes règles R5 et R8. Elle s'applique sous les conditions suivantes : (1) les nœuds de p_2 ont une seule arête de branche principale rentrante et sortante, (2) Q est un /-prédicat tel que sa présence sur n vérifie la condition des *extended skeletons*, (3) pour tous les *mappings* ψ de p_2 dans p_1, d' étant le DAG obtenu en unifiant chaque nœud $n' \in p_2$ avec $\psi(n')$, *pattern*$(\lambda_d(n)[Q])$ a un *mapping racine* dans $\text{SP}_{d'}(n)$.



Des algorithmes de réécriture en utilisant APPLY-RULES

Notre nouvelle version de APPLY-RULES préserve l'importante propriété de sa version antérieure, celle de *produire un DAG équivalent* en sortie. Elle peut donc être utilisée pour produire une version correcte et complète de REWRITE en *remplaçant le test d'équivalence DAG-tree* par une application de APPLY-RULES et ensuite de DOMINANT_INTERLEAVING, pour tester par la suite l'équivalence entre le résultat de DOMINANT_INTERLEAVING et la requête initiale. On rappelle que le test d'équivalence entre deux *tree patterns* est une opération efficace car réalisable en temps polynomial.

D'un autre coté, nous montrons dans cette thèse que notre version modifiée de APPLY-RULES a une **complexité polynomiale**. Alors que suite aux résultats de complexité du problème

de réécriture on ne peut pas espérer à la complétude polynomiale dans le cas général, notre version de APPLY-RULES peut être utilisée pour obtenir un algorithme *correct* de réécriture, comme suggéré dans [16], mais qui soit en plus *polynomial* :

EFFICIENT-RW$(q, \mathcal{V})$
1 **for** p un préfixe sans perte de *pattern*(q)
2 **do**
3 $r \leftarrow$ BUILDINITREWITECANDIDATE$(p, \mathcal{V})$
4 $d \leftarrow$ *dag*(*unfold*(r))
5 $p_1 \leftarrow$ APPLY-RULES(d) ;
6 **if** p_1 est un arbre et $p_1 \equiv p$
7 **then return** compensate$(r, q,$ OUT$(p))$
8 **return** $\emptyset$

Complétude. Dans [16] et [56] on montre que cet algorithme est également *complet* pour des requêtes et des vues dans le fragment $XP_{//}$, *quand les plans considères intersectent des requêtes akin* (c'est-à-dire, qui commencent par la même /-séquence, aussi appelée *token*). Dans cette thèse on montre également une version de cet algorithme qui est **complète pour résoudre le problème de réécriture pour des requêtes dans XP_{es} et des vues dans *XP***. On utilise dans la description ci-dessous la notation $s(d)$ pour designer la forme *extended skeleton* d'un *pattern* d, qui consiste à y enlever les prédicats qui ne respectent pas la condition XP_{es}.

EFFICIENT-RW$(q, \mathcal{V})$
1 **for** p a lossless prefix of *pattern*(q)
2 **do**
3 $r \leftarrow$ BUILDINITREWITECANDIDATE$(p, \mathcal{V})$
4 $d \leftarrow$ *dag*(*unfold*(r))
5 $p_1 \leftarrow$ APPLY-RULES(**s(d)**) ;
6 **if** p_1 est un arbre et $p_1 \sqsubseteq p$
7 **then return** compensate$(r, q,$ OUT$(p))$
8 **return** $\emptyset$

Evaluation experimentale

La motivation principale du travail présenté dans le deuxième chapitre de cette thèse a été celle d'atteindre des performances significatives sur le plan pratique. Ceci a conduit aux améliorations de APPLY-RULES pour atteindre une complexité polynomiale, mais également à une gamme importante d'optimisations sur l'application des règles, présentées en détail dans le manuscrit. Nous avons intégré ces optimisation dans une implémentation complète des algorithmes de réécriture (REWRITE , EFFICIENT-RW et sa version pour le fragment XP_{es}), évaluée ensuite comme suit :

- Nous avons généré avec XMark trois documents XML, de tailles 41KB, 91MB et 18GB

- Nous avons construit notre propre générateur de requêtes et de vues pour un document donné, que nous avons ensuite utilisé pour produire des requêtes appartenant aux trois fragments XPath analysés (en XP_{es}, en $XP_{//}$ sans être en XP_{es}, en XP sans être en $XP_{//}$), de longueur de branche principale 5, 7 et 9 (la profondeur des documents XMark est 11), avec pour chaque couple catégorie et longueur 10 requêtes générées. Pour chaque requête on a généré des ensembles de vues de taille 40, 80, 160, 320, et 640, dont 10% qui ont un *mapping* dans la requête (sont donc incluses dans au moins un plan candidat) sans en être équivalentes (pour ainsi cibler uniquement des réécritures comprenant plusieurs vues).

REWRITE **vs.** EFFICIENT-RW . Le premier aspect investigué a été celui de comparer l'algorithme complet mais pas performant REWRITE et sa version polynomiale EFFICIENT-RW qui n'est complète que dans certaines conditions. Nous avons testé ces algorithmes sur 300 configurations pour des requêtes dans les fragments $XP_{//}$ et XP, en imposant un *timeout* de 30 minutes pour REWRITE . Le résultat de cette évaluation est très intéressant : dans aucun des cas testés nous n'avons obtenu une réécriture en passant *uniquement par* DOMINANT_INTERLEAVING (qui, nous le rappelons, constitue la différence coûteuse entre les deux algorithmes). Dans un tiers des cas analysés le *timeout* a été atteint sans terminaison de REWRITE . Nous pouvons en déduire que (i) la pratique confirme que la complétude de EFFICIENT-RW s'étend *outre les conditions théoriques*, mais aussi que (ii) dans les situations où cela peut ne pas être le cas (celles où le *timeout* a été atteint sans conclusion), le temps passé dans le calcul des *interleavings* est d'un ordre tel que ce calcul *perd tout intérêt pratique*.

Temps de réécriture et evaluation. Nous avons continué notre évaluation en nous focalisant sur des requêtes et des ensembles de vues qui fournissent une réécriture par EFFICIENT-RW et nous avons mesuré les performances de réécriture ainsi que le gain global en évaluation. Les mesures du temps que prend EFFICIENT-RW pour trouver des réécritures sont illustrées ci-dessous, pour les ensembles de requêtes et de vues mentionnées précédemment, générées à partir du document de taille 91MB. Pour les comparaisons de temps d'évaluation, nous avons retenu une requête de taille maximale (9) et ses versions XP_{es} et $XP_{//}$, et nous avons comparé, sur les trois documents, le temps cumulé pour le calcul de la réécriture et son évaluation sur les vues, au temps d'évaluation de la requête initiale. L'ensemble des mesures est présenté ci-dessous.

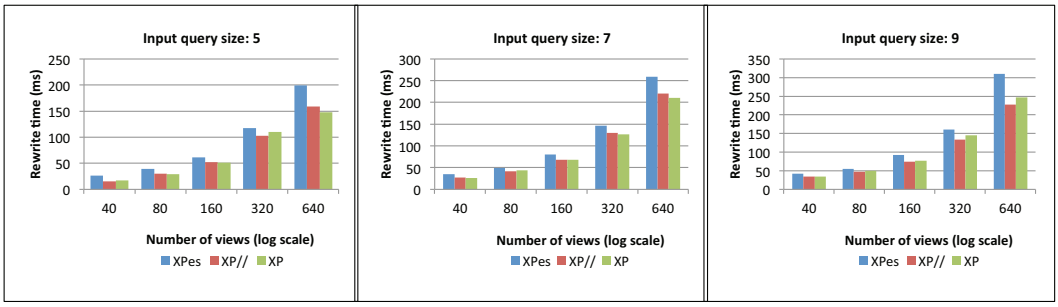


FIGURE B.6: Temps de réécriture

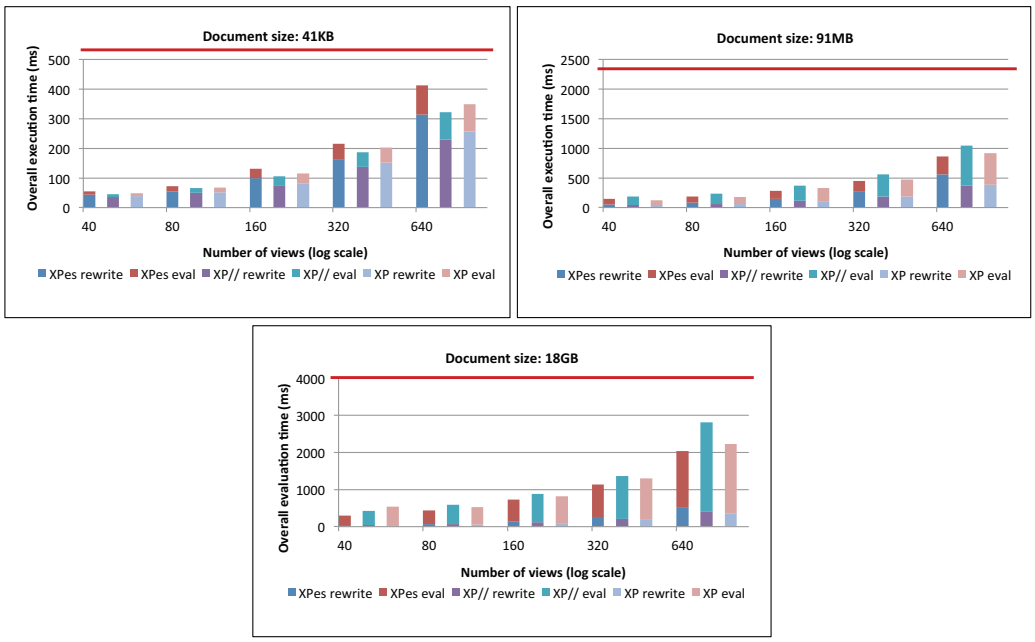


FIGURE B.7: Comparaison des temps d'evaluation.

Discussion. Notre évaluation expérimentale nous permet de constater que notre implémentation de la version modifiée et optimisée de EFFICIENT-RW exhibe des performances significatives sur le plan pratique, et peut passer à l'échelle pour des ensembles de vues de taille importante. On remarque également le gain de performance important en utilisant une réécriture et une évaluation sur les vues. Enfin, EFFICIENT-RW se remarque par son intérêt pratique pour le fragment *XP*, malgré sa complétude théorique limitée, en fournissant une solution rapide et bénéfique pour les scénarios de réécriture en général.

B.4 Conclusions et futures directions de recherche

Le premier et principal chapitre de cette thèse présente une approche globale et efficace pour trouver toutes les reformulations minimales d'une requête relationnelle conjonctive, sous des contraintes d'intégrité. Nous y montrons notre nouvel algorithme, correct et complet, appelé le $Prov_{C\&B}$ (Provenance-Aware Chase & Backchase), et nous présentons sa caractérisation théorique détaillée. Avec notre algorithme, nous introduisons un nouveau type de technique de *chase*, la Provenance-Aware Chase, et sa technique sous-jacente appelée la Conservative Chase, et nous montrons comment les annotations de provenance nous permettent de retrouver directement les reformulations minimales. Nous montrons également comment notre algorithme peut être adapté pour trouver de façon plus directe et plus performante les reformulations de coût minimum.

Nous estimons qu'il serait intéressant de continuer l'investigation théorique des nouvelles techniques de *chase* introduites dans cette thèse, et de leur similarité ou divergence par rapport à la *chase* standard. Une direction très prometteuse sur le plan théorique est aussi celle concernant l'adaptation de la Provenance-Aware Chase à d'autres types de provenance, par exemple pour les bases de données probabilistes.

Sur un plan pratique, nous montrons les performances de notre algorithme et de notre implémentation, pouvant induire des gains jusqu'à deux ordres de magnitude par rapport à un SGBD commercial. Il existe toutefois un potentiel important d'optimisation qui reste encore à exploiter sur le plan pratique, notamment concernant les structures de données efficaces pour la gestion des formules booléennes, ainsi que la construction des plans pour le prémisses des contraintes. Nous allons explorer ces directions dans nos prochaines implémentations, et élargir l'utilisation de notre algorithmes aux nombreux scénarios de reformulation de requêtes rencontrés en pratique.

Dans le second chapitre de cette thèse, nous revisitons et enrichissons le travail de Cautis, Deutsch et Onose, présenté dans [16] et détaillé dans [56], sur la réécriture de requêtes XPath avec un seul niveau d'intersection de plusieurs vues. Nous développons l'analyse de ce problème en montrant ses connections avec le problème de *l'équivalence DAG-tree* et le problème de la *union-freeness d'un DAG*.

Notre principale motivation étant celle d'atteindre des performances pratiques, nous raffinons l'algorithme à base de règles proposé par [16] pour assurer sa complexité polynomiale et améliorer sa complétude. Nous fournissons une implémentation des algorithmes de réécriture, et son évaluation expérimentale extensive. Pour atteindre les performances exhibées, nous présentons également un ensemble important d'optimisations, à la frontière de la théorie et la pratique. Même si nos résultats expérimentaux sont très prometteurs, le potentiel d'optimisation des règles reste important, et nous estimons qu'une étude théorique de l'impact et de l'interaction de ces règles permettrait de raffiner leur caractérisation et d'améliorer encore les performances. Ceci pourrait aussi servir à définir une frontière de tractabilité et son extension à des plans de réécriture plus complexes.

Comme suggéré par le titre de cette thèse, les deux chapitres approchent le problème de la réécriture des requêtes avec des vues et de la reformulation des requêtes, d'une perspective en égale mesure théorique et pratique. Dans notre vision, ces deux points de vue doivent être entrelacés et se renforcer constamment pour acquérir des résultats significatifs. Nous pensons aussi que, même s'ils ont été depuis longtemps traités dans la recherche en bases de données, les sujets analysés dans cette thèse seront constamment remis à jour par des scénarios applicatifs nouveaux, plus complexes et à plus grande échelle, fournissant ainsi de nouvelles opportunités et défis d'optimisation théorique et pratique.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] F. Afrati, M. Damigos, and M. Gergatsoulis. Union rewritings for XPath fragments. In *Proceedings of the 15th Symposium on International Database Engineering*, IDEAS '11, pages 43–51, 2011.

[3] F. N. Afrati, R. Chirkova, M. Gergatsoulis, and V. Pavlaki. Finding equivalent rewritings in the presence of arithmetic comparisons. In *EDBT*, pages 942–960, 2006.

[4] F. N. Afrati, C. Li, and P. Mitra. Answering queries using views with arithmetic comparisons. In *PODS*, 2002.

[5] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *SIGMOD Conference*, 2003.

[6] P. Aravogliadis and V. Vassalos. On equivalence and rewriting of XPath queries using views under DTD constraints. In *DEXA (2)*, pages 1–16, 2011.

[7] A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, pages 60–71, 2004.

[8] R. G. Bello, K. Dias, A. Downing, J. Feenan, J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *VLDB*, pages 659–664, 1998.

[9] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS*, pages 25–36, 2005.

[10] M. Benedikt, W. Fan, and G. Kuper. Structural properties of XPath fragments. *Theor. Comput. Sci.*, 336(1):3–31, 2005.

[11] M. Benedikt, B. ten Cate, and E. Tsamoura. Generating low-cost plans from proofs. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, 2014.

[12] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0, 2007.

[13] S. Boag, D. Chamberlain, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, 2007.

[14] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[15] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma. Extracting content structure for web pages based on visual representation. APWeb, 2003.

[16] B. Cautis, A. Deutsch, and N. Onose. Xpath rewriting using multiple views: Achieving completeness and efficiency. In *11th International Workshop on the Web and Databases, WebDB 2008, Vancouver, BC, Canada, June 13, 2008*, 2008.

[17] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[18] L. Chen and E. A. Rundensteiner. XCache: XQuery-based caching system. In *WebDB*, pages 31–36, 2002.

[19] L. J. Chen and Y. Papakonstantinou. Supporting top-k keyword search in xml databases. In *ICDE*, 2010.

[20] J. Clark and S. DeRose. XML path language (XPath), 1999.

[21] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM TODS*, 31(2), 2006.

[22] V. Crescenzi, G. Mecca, and P. Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *VLDB*, 2001.

[23] D. DeHaan. Equivalence of nested queries with mixed semantics. In *PODS*, pages 207–216, 2009.

[24] N. Derouiche. Recherche des objets complexes dans le web structuré. 2012.

[25] N. Derouiche, B. Cautis, and T. Abdessalem. Automatic extraction of structured web data with domain knowledge. In *ICDE*, pages 726–737, 2012.

[26] A. Deutsch and R. Hull. Provenance-directed chase&backchase. In *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, 2013.

[27] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, 1999.

[28] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[29] A. Deutsch and V. Tannen. Mars: A system for publishing xml from mixed and redundant storage. In *VLDB*, pages 201–212, 2003.

[30] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, pages 201–212, 2003.

[31] A. Deutsch and V. Tannen. Reformulation of XML queries and constraints. In *ICDT*, 2003.

[32] R. Fagin. Horn clauses and database dependencies. *J. ACM*, 1982.

[33] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[34] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular XPath queries on XML views. In *ICDE*, pages 666–675, 2007.

[35] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, pages 311–322, 1999.

[36] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, 2001.

[37] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive queries over trees. In *PODS*, 2004.

[38] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[39] S. Groppe, S. Böttcher, and J. Groppe. XPath query simplification with regard to the elimination of intersect and except operators. In *ICDE Workshops*, page 86, 2006.

[40] A. Halevy. Answering queries using views: A survey. *VLDB J.*, 2001.

[41] J. Hidders. Satisfiability of XPath expressions. In *DBPL*, pages 21–36, 2003.

[42] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete yet practical search for minimal query reformulations under constraints. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014.

[43] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-based data pricing. In *PODS*, pages 167–178, 2012.

[44] P.-Å. Larson, J. Goldstein, H. Guo, and J. Zhou. MTCaches: Mid-tier database caching for SQL Server. *IEEE Data Eng. Bull.*, 27(2), 2004.

[45] M. Levene and G. Loizou. Why is the snowflake schema a good data warehouse design? *Information Systems*, 28(3):225 – 240, 2003.

[46] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[47] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external query processors. *J. Comput. Syst. Sci.*, 1999.

[48] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, pages 469–480, 2005.

[49] I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, pages 972–983, 2011.

[50] B. Marnette. Generalized schema-mappings: from termination to tractability. In *PODS*, pages 13–22, 2009.

[51] B. Marnette and F. Geerts. Static analysis of schema-mappings ensuring oblivious termination. In *ICDT*, pages 183–195, 2010.

[52] M. Meier, M. Schmidt, and G. Lausen. On chase termination beyond stratification. *PVLDB*, 2(1):970–981, 2009.

[53] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[54] A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *ICDE*, 1989.

[55] A. Nash and B. Ludäscher. Processing first-order queries under limited access patterns. In *PODS*, 2004.

[56] N. Onose. Uncovering the full potential of data services. 2009.

[57] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested xml queries using nested views. In *SIGMOD*, 2006.

[58] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, pages 443–454, 2006.

[59] L. Popa. *Object/relational Query Optimization with Chase and Backchase*. PhD thesis, University of Pennsylvania, 2000.

[60] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far? In *SIGMOD*, pages 273–284, 2000.

[61] S. Rizvi, A. O. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, pages 551–562, 2004.

[62] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XML path language (XPath) 3.0, 2010.

[63] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.

[64] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *VLDB*, 1996.

[65] J. Tang and S. Zhou. A theoretic framework for answering XPath queries using views. In *XSym*, pages 18–33, 2005.

[66] B. ten Cate and C. Lutz. The complexity of query containment in expressive fragments of XPath 2.0. In *PODS*, pages 73–82, 2007.

[67] X. Wu, D. Theodoratos, and W. H. Wang. Answering XML queries using materialized views revisited. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM, pages 475–484, 2009.

[68] W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, pages 121–132, 2005.

[69] L. H. Yang, M. L. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. In *VLDB*, pages 69–80, 2003.

[70] C. Yu and L. Popa. Constraint-based xml query rewriting for data integration. In *SIGMOD*, pages 371–382, 2004.

[71] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *SIGMOD*, pages 105–116, 2000.

[72] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *WWW*, 2005.