# Complete Yet Practical Search for Minimal Query Reformulations Under Constraints

Ioana Ileana, Bogdan Cautis, Alin Deutsch, Yannis Katsis

# Complete Yet Practical Search for Minimal Query Reformulations Under Constraints

Ioana Ileana
Institut Mines-Télécom; Télécom ParisTech; CNRS LTCI
ileana@telecom-paristech.fr

Bogdan Cautis
Univ. Paris-Sud & Inria Saclay
bogdan.cautis@u-psud.fr

Alin Deutsch
University of California San Diego
deutsch@cs.ucsd.edu

Yannis Katsis
University of California San Diego
ikatsis@cs.ucsd.edu

## ABSTRACT

We revisit the Chase&Backchase (*C&B*) algorithm for query reformulation under constraints, which provides a uniform solution to such particular-case problems as view-based rewriting under constraints, semantic query optimization, and physical access path selection in query optimization. For an important class of queries and constraints, *C&B* has been shown to be complete, i.e. guaranteed to find all (join-)minimal reformulations under constraints. *C&B* is based on constructing a canonical rewriting candidate called a universal plan, then inspecting its exponentially many sub-queries in search for minimal reformulations, essentially removing redundant joins in all possible ways. This inspection involves chasing the subquery. Because of the resulting exponentially many chases, the conventional wisdom has held that completeness is a concept of mainly theoretical interest.

We show that completeness can be preserved at practically relevant cost by introducing $Prov_{C\&B}$, a novel reformulation algorithm that instruments the chase to maintain provenance information connecting the joins added during the chase to the universal plan subqueries responsible for adding these joins. This allows it to directly "read off" the minimal reformulations from the result of a single chase of the universal plan, saving exponentially many chases of its subqueries. We exhibit natural scenarios yielding speedups of over two orders of magnitude between the execution of the best view-based rewriting found by a commercial query optimizer and that of the best rewriting found by $Prov_{C\&B}$ (which the optimizer misses because of limited reasoning about constraints).

## Categories and Subject Descriptors

H.2 [**Database Management**]: Systems—*Query processing*

## Keywords

database views, integrity constraints, query optimization, chase

## 1. INTRODUCTION

One of the central problems in query processing is that of reformulating a query $Q$ expressed against a source schema $S$ to an equivalent query $R$ against a target schema $T$, by exploiting the relationship between $S$ and $T$. In general, there may exist multiple such reformulations and choosing among them may require additional, application-specific criteria, such as a cost model.

Query reformulation includes as particular cases several problems that have occupied database research and practice for decades. Examples are view-based rewriting (both partial and total, in which base tables are allowed, respectively disallowed), physical access path selection in query optimization, and semantic optimization (e.g. redundant join elimination and other instances of rewriting queries under integrity constraints).

Motivated by the fact that these particular case problems had been typically studied in isolation from each other in prior work, publication [10] introduced a uniform solution in the form of the Chase & Backchase (*C&B*) algorithm for query reformulation under constraints. The paper showed how the above problems can be cast as particular query reformulation instances where the relationship between the schemas $S$ and $T$ is expressed by constraints.

The *C&B* algorithm applies to relational conjunctive queries (select - project - join - rename under set semantics), as the language for specifying the original query $Q$ and the reformulation $R$. The relationship between the schemas $S$ and $T$ is given by a set $\mathcal{C}$ of constraints that are expressed as *tuple-generating dependencies (tgds)* or *equality-generating dependencies (egds)* [4]. Tgds and egds together comprise the embedded implicational dependencies [14], which 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.). Furthermore, tgds turn out to be ideally suited for capturing physical access paths typically used in query optimization (materialized views expressed as SPJR queries, indexes, join indexes, access support relations, gmaps, etc.) [10].

In a nutshell, the *C&B* is based on constructing a canonical rewriting 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 joins that are redundant under the constraints. The inspected subqueries are checked for equivalence (under the constraints) to the original query. This check is performed using the classical chase procedure [1], which in essence adds to a query redundant joins that are implied by the constraints.

The *C&B* was shown in [10] to discover at least the reformulations found by prior techniques in the particular case problems, and potentially more. More importantly, the *C&B* was shown experimentally to uncover previously missed reformulation opportunities

Supported by INRIA (as OAKSAD associated team) and by the NSF (awards IIS-1117527, -0910820 and CNS-1219220).


*SIGMOD'14,* June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 http://dx.doi.org/10.1145/2588555.2593683 $15.00.

stemming from the synergistic combination of opportunities across the settings of the particular case problems. Examples include cases when the integrity constraints are crucial in enabling the use of certain indexes or views, in the sense that the corresponding reformulation that mentions these indexes or views is not equivalent to the original query in the absence of the integrity constraints, but becomes so in their presence [29]. This *C&B* behavior turned out to have a theoretical underpinning: in [13], the *C&B* algorithm was proven to return all equivalent minimal reformulations of a query. The *C&B* is said to be *complete* for finding minimal reformulations.

Completeness is desirable for two reasons. First, there are modern applications (discussed in Section 7) that define a certain measure of a query (e.g. its price or clearance level) as the overall minimum across *all* its reformulations. Second, even in classical optimization 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, our experiments show that 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 rewriting of a query has an NP-hard decision problem even in the absence of constraints [22], conventional wisdom has held so far that completeness of reformulation is likely to remain a concept of mainly theoretical interest. Indeed for the *C&B*, 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. [29] 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, even when restricted to reformulating conjunctive queries within the settings of the particular case problems discussed here. This includes the optimizers of relational DBMSs and the follow-up *C&B*-based implementations for XML query reformulation in [12, 27, 32].

**Our contributions.** In this work, we revisit the *C&B* and we challenge conventional wisdom by showing that the cost of running a complete search for minimal reformulations can be reduced in practice to a small fraction of typical query execution times, while the benefits are potentially huge in certain practically relevant settings. This result is enabled by the following specific contributions.

**1.** We present $Prov_{C\&B}$, a query reformulation algorithm that 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*. We formally prove that the $Prov_{C\&B}$ is still complete, thus always finding precisely the same minimal reformulations as the *C&B* (namely all existing ones). The $Prov_{C\&B}$'s search is based on a novel *provenance-aware chase*, which tracks provenance information that serves for tracing the added joins back to the universal plan subqueries responsible for the joins 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 *C&B* would perform). The reduction in the number of chases is the fundamental reason for the speedup of $Prov_{C\&B}$ over *C&B*. Remarkably, the particular provenance flavor required turns out to coincide with *minimal why-provenance* [6] introduced for a different purpose in the literature.

**2.** We evaluate $Prov_{C\&B}$ experimentally in a setting involving materialized views and integrity constraints. We confirm experimentally that the savings of $Prov_{C\&B}$ over *C&B* in terms of launched chases can be exponential. The experiments also show that the running time of $Prov_{C\&B}$ is a very small fraction of the query execution time measured for modern commercial database engines. Moreover, they show that running the $Prov_{C\&B}$ is worthwhile: we exhibit scenarios with speedups of over two orders of magnitude between the execution of the best view-based rewriting found by a commercial query optimizer and a rewriting obtained by selecting a minimum-join rewriting among all minimal rewritings enumerated by $Prov_{C\&B}$. The reason is that the optimizer misses the opportunity to exploit available materialized views because it does not realize that they are rendered relevant to the query by the declared key (and sometimes foreign key) constraints. The observed speedups take into account the combined time spent by the $Prov_{C\&B}$ to select a rewriting, and the time to execute it. Finally, in the same materialized view scenario, we evaluate how $Prov_{C\&B}$ scales with the size of the query and combinatoric explosion of available rewritings, showing graceful behavior even in stress-test settings that go significantly beyond what existing commercial optimizers are designed to handle.

**3.** The provenance-aware chase is interesting in its own right, as a procedure for reasoning about the interaction of provenance and integrity constraints. 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 soundness of the resulting reformulation algorithm (i.e. it would return non-equivalent reformulations). We were forced to first design a new (provenance-unaware) variation of the standard chase, which we call the *Skolem chase*, showing that its results are equivalent to those of the standard chase. It is this variation that we instrumented to obtain the provenance-aware chase.

**Beyond relational conjunctive queries.** The original *C&B* algorithm has been extended in follow-up work to apply beyond conjunctive queries (see [11] for a survey of these extensions). The extensions allow disjunction/union [13], 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 [27, 32]. Moreover, extensions support additional data models, such as object-oriented, complex-valued [10] and XML [12, 13, 32]. 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 behavior of the *C&B* (shared by the $Prov_{C\&B}$) and not on its internal working.

**Applications of the $Prov_{C\&B}$.** The $Prov_{C\&B}$ algorithm is applicable in a myriad of well-known scenarios involving query reformulation. However, we single out a class of applications that are particularly well-suited for the $Prov_{C\&B}$: they need the optimum among all minimal reformulations, of which there are rela-

tively few, but these are hard to find because the search space for them is large. We discuss this class in Section 7.

**Paper organization.** We recall the *C&B* in Section 2, then present the $Prov_{C\&B}$ in Section 3, its implementation in Section 4 and its evaluation in Section 5. We discuss related work in Section 6 and conclude in Section 7.

## 2. OVERVIEW OF *C&B*

We recall the *C&B* algorithm, starting with the problem of query reformulation and continuing with an illustration of the *C&B*. For presentation simplicity, we set the illustration in a very restricted case of query reformulation, namely total rewriting of queries using materialized views in the absence of integrity constraints. Examples including integrity constraints are given in Section 5.

**Reformulation.** We write $D \models \mathcal{C}$ if a database instance $D$ satisfies all the constraints in a set $\mathcal{C}$. 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 $Q_2 \sqsubseteq_{\mathcal{C}} Q_1$. Let $S$ and $T$ be relational schemas related by set $\mathcal{C}$ of constraints, and $Q$ a query formulated against $S$. A *$T$-reformulation* of $Q$ under $\mathcal{C}$ is a query $R$ formulated against $T$, such that $Q \equiv_{\mathcal{C}} R$.

**Minimal reformulation.** 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), and constraints expressed as embedded dependencies [1]. In this context, a query is *$\mathcal{C}$-minimal* if it contains no joins that are redundant under the constraints $\mathcal{C}$, i.e. no join can be removed while preserving equivalence (under $\mathcal{C}$) to the original query.

**The *C&B* algorithm.** We illustrate the *C&B* on the following running example.

EXAMPLE 2.1. *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*[1]

$Q$ : *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 in products. It is easy to see that*

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

$R_1$: *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 joins can be removed while preserving equivalence to $Q$.*

*The* C&B *algorithm casts this rewriting problem as the following instance of the reformulation problem. The source schema is the logical schema against which query $Q$ is formulated (tables $R$, $S$, and $T$ in this example). 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 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. $\mathcal{C}_\mathcal{V}$ is obtained canonically by stating the inclusion (in both directions) between the result of the query defining each view and the view's extent. For the example, $\mathcal{C}_\mathcal{V}$ is the following set of constraints (they are all tgds, and thus embedded dependencies):*

$$c_{V_R}\colon\ \forall r, r \in R \rightarrow \exists v_r, v_r \in V_R \wedge v_r.A = r.A \wedge v_r.C = r.C$$
$$b_{V_R}\colon\ \forall v_r, v_r \in V_R \rightarrow \exists r, r \in R \wedge r.A = v_r.A \wedge r.C = v_r.C$$
$$c_{V_S}\colon\ \forall s, s \in S \rightarrow \exists v_s, v_s \in V_S \wedge v_s.C = s.C \wedge v_s.D = s.D$$
$$b_{V_S}\colon\ \forall v_s, v_s \in V_S \rightarrow \exists s, s \in S \wedge s.C = v_s.C \wedge s.D = v_s.D$$
$$c_{V_{RS}}\colon \forall r, s,\ r \in R \wedge s \in S \wedge r.C = s.C \rightarrow \exists v_{rs}, v_{rs} \in V_{RS} \wedge v_{rs}.A = r.A \wedge v_{rs}.D = s.D$$
$$b_{V_{RS}}\colon \forall v_{rs}, v_{rs} \in V_{RS} \rightarrow \exists r, s,\ r \in R \wedge s \in S \wedge r.A = v_{rs}.A \wedge s.D = v_{rs}.D \wedge r.C = s.C$$
$$c_{V_T}\colon\ \forall t, t \in T \rightarrow \exists v_t, v_t \in V_T \wedge v_t.D = t.D \wedge v_t.E = t.E$$
$$b_{V_T}\colon\ \forall v_t, v_t \in V_T \rightarrow \exists t, t \in T \wedge t.D = v_t.D \wedge t.E = v_t.E$$

The *C&B* algorithm relies on the *chase* procedure, which essentially adds to a query the redundant joins 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 2.1) 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 2.1) 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 $q$, 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 [7]. The extension of $q$ 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 all these atoms already exist in the query, then the chase step is said to *not apply*, and it turns into a no-op.

EXAMPLE 2.2. *We illustrate a chase step of query $Q$ from Example 2.1 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 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 $Q^{\mathcal{C}}$.[2]

The *C&B* algorithm proceeds in two phases:

*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 dropping all joins with relations from schema $S$.

*Backchase:* This phase checks the subqueries of the universal plan $U$ for equivalence (under $\mathcal{C}$) to $Q$. Informally, a *subquery* of $U$ is obtained by essentially selecting a subset $s$ of $U$'s tuple variables and keeping only the atoms involving these variables (we say that the subquery is *induced* by $s$). The minimal equivalent subqueries are returned. Minimal subqueries contain no redundant joins under $\mathcal{C}$, i.e. no joins whose removal would induce a subquery that is itself equivalent to $Q$ under $\mathcal{C}$. 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 $sq^{\mathcal{C}}$.

EXAMPLE 2.3. 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*[3]

$U$*: 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 $\{v_{rs}, v_t\}$.*

*We illustrate only for $R_2$. To show that $R_2$ is equivalent to Q, the* C&B *chases $R_2$ with $\mathcal{C}_{\mathcal{V}}$ and searches for a containment mapping from Q into $R_2^{\mathcal{C}_{\mathcal{V}}}$. The only applicable chase steps involve $b_{V_{RS}}, b_{V_T}$, yielding the result*

$R_2^{\mathcal{C}_{\mathcal{V}}}$*: 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 $R_2^{\mathcal{C}_{\mathcal{V}}}$, $R_2$ 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, 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 the join with $v_s$ is redundant.*

**Completeness of the *C&B*.** The fact that rewritings $R_1$ and $R_2$ in Example 2.1 are discovered among the subqueries of $U$ is not accidental. In [10], it was shown that all minimal rewritings of $Q$ are (isomorphic to) subqueries of $U$, in the absence of integrity constraints. The result was extended to the presence of integrity constraints expressed as embedded dependencies as long as the chase with them terminates [13].

**Chase termination** For arbitrary sets $\mathcal{C}$ of constraints, the chase procedure is not guaranteed to terminate. One of the least restrictive (and the 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* [15].

**Relevant *C&B* implementation details** The first *C&B* implementation is described in [29], where the backchase phase is identified as the performance bottleneck. This is expected, since exponentially many subqueries of the universal plan are checked for equivalence with the original query, and each equivalence check involves a chase. While [10] shows that this brute-force search is optimal from a complexity-theoretic point of view, follow-up work concerns itself with practical feasibility and proposes techniques for pruning the search. The only completeness-preserving pruning technique, sketched in [29] and detailed in [28], boils down to enumerating subqueries of the universal plan $U$ in a bottom-up fashion, starting with all single-atom subqueries, next with two-atom subqueries, etc. Since the backchase searches for minimal reformulations, this bottom-up strategy allows pruning the equivalence check for all subqueries *sq* of $U$ that already include as subquery a reformulation, since all such *sq* are non-minimal. In Example 2.3, subquery $sq'$ would be pruned this way.

Even with bottom-up pruning, exponentially many subqueries remain to be chased in the worst case. In practice, this worst case occurs often. For instance, if the minimum number of joins in a reformulation is $N$, all subqueries of $U$ up to size $N$ are inspected since the pruning never kicks in for them. Their number is exponential in $N$. In the particular case when we seek total view-based rewritings of a query and none exist, all possible subqueries of $U$ need to be checked. To at least mitigate this case, [29] proposes first checking that $Q$ has a rewriting before even starting the subquery enumeration. This check is performed as follows.

A corollary of the completeness of the *C&B* algorithm states that $Q$ has a reformulation if and only if it has a containment mapping into $U^{\mathcal{C}}$, i.e. into the result of chasing the universal plan $U$ with the dependencies in $\mathcal{C} = \mathcal{C}_{\mathcal{V}} \cup \mathcal{C}_{\mathcal{I}}$. In practical implementations (e.g. in [29]), the existence of a containment mapping from $Q$ into $U^{\mathcal{C}}$ is checked by treating $U^{\mathcal{C}}$ as a small symbolic database instance (known as "canonical" instance in the literature [1]), and evaluating $Q$ over it. This amounts to computing the set of *all* containment mappings from $Q$ into $U^{\mathcal{C}}$, and checking its non-emptiness.

[2] 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 in the remainder of this paper.

[3] Equalities of terms involving view variables that were implicit in $Q^{\mathcal{C}_{\mathcal{V}}}$ are made explicit in $U$.

EXAMPLE 2.4. *Revisiting Example 2.3, 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* $t.E = v_t.E$

*If we evaluate $Q$ over the canonical instance of $U^{\mathcal{C}_\mathcal{V}}$, we obtain the containment mappings $h_1 = \{r \mapsto r_1, s \mapsto s_1, t \mapsto t\}$ and $h_2 = \{r \mapsto r_2, s \mapsto s_2, t \mapsto t\}$. Therefore, $U$ is a (redundant) rewriting of $Q$, and it makes sense to start inspecting its subqueries in search of minimal rewritings.*

# 3. A NOVEL ALGORITHM: $Prov_{C\&B}$

The remainder of this paper shows that significantly more can be done to prune the search for minimal reformulations while preserving completeness.

Intuitively, the original backchase enumerates the subqueries of the universal plan $U$ in a bottom-up fashion and chases each of them in isolation from the others, to determine equivalence to the input query $Q$. This leads to *redundant chasing* of the atoms occurring in common within distinct subqueries of $U$. It also leads to *fruitless chasing* for subqueries that are not equivalent to $Q$.

EXAMPLE 3.1. *In our running example, the bottom-up backchase search prunes all strict superqueries of $R_1, R_2$ except $U$. This leads to pruning subqueries $v_r \wedge v_{rs} \wedge v_t$ and $v_s \wedge v_{rs} \wedge v_t$.*[4]

*In addition, the backchase will prune those subqueries that do not contain the universal plan's head variables, as only safe rewritings are of interest (e.g. it will prune $v_s$, and $v_s \wedge v_t$).*

*However, the backchase still carries out fruitless chases of the following 7 subqueries of $U$, only to determine that none of them are rewritings of $Q$: $v_r$, $v_{rs}$, $v_r \wedge v_s$, $v_r \wedge v_t$, $v_r \wedge v_{rs}$, $v_s \wedge v_{rs}$, $v_r \wedge v_s \wedge v_{rs}$. Notice how the common relational atom $V_T\ v_t$ is chased redundantly multiple times (once when chasing $U$, then again when chasing $R_1$, and also when chasing $R_2$, and in the fruitless chases of the three above-listed subqueries involving $v_t$).*

*One might wonder why the backchase won't more aggressively prune away all subqueries whose combined views don't even mention all relations mentioned by $Q$. In our example, this would immediately dismiss all 7 subqueries listed above. Note that this aggressive pruning is unsafe in general, in the sense of compromising completeness of the backchase. Indeed, if the set of constraints includes tuple-generating dependencies (such as foreign key constraints), the minimal rewritings do not necessarily mention all relations mentioned by the query. It is easy to construct such examples: for instance, assume that $S.D$ is a foreign key referencing $T.D$. Then subqueries $v_r \wedge v_s$ and $v_{rs}$ are minimal rewritings that would be missed by the aggressive pruning.*

Our aim is to avoid both fruitless and redundant chasing, by replacing the many isolated subquery chases with a single chase of the universal plan $U$. To this end, we propose a new backchase strategy that keeps track of the provenance of each atom it creates, where the *provenance* of an atom $a$ gives the set of subqueries of $U$ whose chasing led to the creation of $a$. This provenance information enables us to run $Q$ over the canonical instance of the result $U'$ of chasing $U$, identify each image $i$ of $Q$ into $U'$, and trace it back to the subqueries of $U$ that are responsible for the creation of $i$ during the chase of $U$.

EXAMPLE 3.2. *We illustrate by revisiting Example 2.4. We show again $U^{\mathcal{C}_\mathcal{V}}$, this time annotating the relational atoms of $U^{\mathcal{C}_\mathcal{V}}$ with their provenance in terms of the view atoms in $U$. The provenance annotations appear in square brackets.*

$U^{\mathcal{C}_\mathcal{V}}$: *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* $t.E = v_t.E$

*Note that the relational atoms in $U^{\mathcal{C}_\mathcal{V}}$ involving views are annotated with themselves, as they are not introduced by chasing, but instead inherited directly from $U$. The variable bindings involving $R, S, T$ are introduced by the chase, for instance $r_1$ by chasing $v_{rs}$ with view dependency $b_{V_{RS}}$ and $r_2$ by chasing $v_r$ with $b_{V_R}$.*

*Recall from Example 2.4 that $Q$ has precisely two containment mapping images into $U^{\mathcal{C}_\mathcal{V}}$: one given by $h_1$, comprising $r_1$, $s_1$ and $t$, and another given by $h_2$, comprising $r_2$, $s_2$ and $t$. The provenance of the first image of $Q$ is $v_{rs} \wedge v_t$, which corresponds to rewriting $R_2$ in Example 2.1, while the provenance of the second image is $v_r \wedge v_s \wedge v_t$, corresponding to rewriting $R_1$.*

*Notice how by computing the containment mappings of $Q$ into $U^{\mathcal{C}_\mathcal{V}}$ (a step that is already carried out in the original* C&B *algorithm), we immediately identify the two rewritings of $Q$, saving the fruitless individual chases of the subqueries listed in Example 3.1.*

*Also notice how the $V_T\ v_t$ atom is only chased once and for all when chasing $U$, saving the redundant chasing that would have resulted from chasing $R_1$ and $R_2$ in isolation (as prescribed by the original backchase).*

**Provenance-Aware Chase (pa-chase)** Recall from Section 2 that the completeness result of the *C&B* algorithm is based on chasing the subqueries of the universal plan $U$ ($U$-subqueries for short). The purpose of the pa-chase is to replace all of these chases with a single chase of the universal plan (and thus of all its subqueries simultaneously), capturing during this chase the *C&B*-relevant effect of the isolated chases of the $U$-subqueries.

Our original motivation in designing the pa-chase was the desire to achieve the following goal:

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

The benefits of such a design would be that (i) by restricting attention to only those universal plan subqueries identified by the provenance annotations we do not miss any 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 this single chase of $U$ sufficient.

The technical challenge facing the implementation of this idea is raised by the need to carefully instrument the chase procedure

[4]To avoid clutter, in the running example we specify universal plan subqueries by mentioning only their tuple variables.

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 instrumentation. We have therefore designed a chase variation we call the Skolem chase. It is provenance-agnostic like the standard chase and equivalent to it in terms of termination behavior and produced result. The Skolem chase is therefore interchangeable with the standard chase within the *C&B* algorithm, and the resulting *C&B*-variation remains complete. The advantage of the Skolem chase is that it lends itself to provenance-tracking instrumentation, yielding the pa-chase which is guaranteed to satisfy the following invariant:

($\diamond$) *The provenance of an atom $a$ constructed during the pa-chase of the universal plan specifies precisely the set of universal plan subqueries whose Skolem chases (conducted in isolation from each other) would construct $a$.*

The pa-chase tracks provenance by annotating each original relational atom of universal plan $U$ with a unique id called a *provenance term*, and by annotating each (relational or equality) atom $a$ introduced by a pa-chase step with a *provenance formula*. Provenance formulae are constructed from provenance terms using logical conjunction and disjunction. A conjunction $c$ of terms specifies a subquery of $U$, namely the subquery $sq$ obtained by restricting $U$'s FROM clause to the relational atoms given by the terms in $c$, and the WHERE clause to the equality atoms involving the corresponding variables. We say that $c$ *induces* $sq$. Intuitively, a conjunction $c$ of terms is intended to express the fact that the Skolem chase of $sq$ constructs $a$. Disjunction is intended to express alternative $U$-subqueries, each of which lead to $a$'s construction when Skolem-chased in isolation.

Before detailing the pa-chase, we show how invariant ($\diamond$) enables the $Prov_{C\&B}$ to replace the *C&B*'s backchase phase.

**Search for Reformulations** Once the universal plan $U$ is pa-chased into result $U'$, we simply compute the set $\mathcal{H}$ of all containment mappings from $Q$ to $U'$ (as done in the standard *C&B*). For each containment mapping $h \in \mathcal{H}$, invariant ($\diamond$) ensures that the provenance formula $\pi(h(Q))$ attached to $Q$'s image under $h$ gives precisely the (possibly multiple) $U$-subqueries whose isolated Skolem chases construct this image, thus witnessing equivalence to $Q$. All of these subqueries are therefore reformulations of $Q$. The reason we mention multiple $U$-subqueries is that in general the provenance formula $\pi(h(Q))$ may contain disjunction, signaling alternative $U$-subqueries each of whose Skolem chases constructs $h(Q)$. These alternative $U$-subqueries are read off directly from the disjunctive normal form (DNF) of $\pi(h(Q))$: every conjunct $c$ corresponds to the $U$-subquery induced by the terms of $c$. The conjuncts of the DNF form of $\bigvee_{h\in\mathcal{H}} \pi(h(Q))$ yield all $U$-subqueries that are reformulations of $Q$. Notice how, due to invariant ($\diamond$), there is no need to check these subqueries for equivalence to $Q$, or to consider any other $U$-subqueries.

**Minimization** We still need to minimize the set of reformulations identified by $\Pi = DNF(\bigvee_{h\in\mathcal{H}} \pi(h(Q))$. 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. All we need to do therefore is to remove from $\Pi$ all subsumed conjuncts, obtaining what we call the *reduced form* of $\Pi$, $rf(\Pi)$. 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.

EXAMPLE 3.3. *Recall Example 3.2. In its simple setting, the standard chase and the Skolem chase behave identically, so there is no need to know the details of the Skolem chase to understand the provenance annotations resulting from pa-chasing $U$ with $\mathcal{C} = \mathcal{C}_\mathcal{V}$ to obtain $U'$. By Example 3.2, we have $\Pi = \pi(h_1(Q)) \vee \pi(h_2(Q)) = v_{rs} \wedge v_t \vee v_r \wedge v_s \wedge v_t$. Notice that $\Pi$ is already in reduced form (DNF with no conjuncts subsumed by others). Each of the two conjuncts of $\Pi$ corresponds to a rewriting of $Q$: the first to $R_2$, the second to $R_1$.*

**Putting It All Together** We summarize $Prov_{C\&B}$ below.

**algorithm** $Prov_{C\&B}$
**params:** source schema $S$, target schema $T$
set $\mathcal{C}$ of weakly-acyclic embedded dependencies over $S \cup T$

**input:** SPJR query $Q$ formulated over $S$,
**output:** all minimal SPJR $T$-reformulations of $Q$ under $\mathcal{C}$

*//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 $\Pi$ as the DNF formula of $\bigvee_{h\in\mathcal{H}} \pi(h(Q))$
5. compute the reduced form $rf(\Pi)$ of $\Pi$
6. return the $U$-subqueries induced by the conjuncts of $rf(\Pi)$.

**Formal guarantees** Our main result shows that the $Prov_{C\&B}$ algorithm preserves completeness:

THEOREM 3.1. *Let $\mathcal{C}$ be a weakly-acyclic set of dependencies. Then for any SPJR query $Q$, the $Prov_{C\&B}$ algorithm returns precisely the minimal reformulations of $Q$ under $\mathcal{C}$.*

**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 whenever the standard chase terminates.

To detail provenance tracking during the pa-chase, we introduce some notation first. Given atom $a$, its provenance formula is $\pi(a)$. The provenance of a set of atoms $A$ is the logical conjunction of the provenances of its members: $\pi(A) = \bigwedge_{a\in A} \pi(a)$.

We detail the intuitions that drove the design of the pa-chase step. Recall that, initially, we were attempting to mimic the behavior of the standard chase, which is why the intuitions below initially refer to a *tentative pa-chase step (tpa-step)* modeled after the standard chase step. In due course, we identify the need to substitute the standard chase with the Skolem chase in the actual definition of pa-chase step, and of 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 chase in isolation creates 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 3.2, 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 premise's homomorphic match $h$ has an extension to a homomorphic match of the conclusion. Denoting the extended match as $h'$, the atoms in $q$ that are "already there" are the atoms in $h'(d_C)$. In designing the pa-chase step, one would be 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 $d$, 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 a 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 behavior 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" ($h'(d_C)$). We illustrate such a scenario in Example 3.4 below. In this case, we need to record the provenance of the atoms in $A$. Where can we record this information? The intuition offered by the standard chase would be to add no new atoms to $q$, 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 $h'(d_C)$. In fact we can give examples showing that if we defined the pa-chase step 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. As a consequence, such a design would require $Prov_{C\&B}$ to check for equivalence each subquery indicated by a provenance annotation, which would involve launching a chase. To avoid this, the tpa-chase step must behave differently than the standard chase step in case *I3*: it adds the atoms $A$ to $q$ and adorns them with provenance $h(d_P)$, while leaving the atoms in $h'(d_C)$ unaffected.

EXAMPLE 3.4. *Recall the pa-chase of universal plan $U$ from Example 3.2, 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 3.2). The intermediate result is $U_2$ below, in which the tuple variables are named to show correspondence to the tuple variables introduced in Example 3.2.*

*$U_2$: 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 3.2). However the standard 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 that $r_2$ is* not *a copy of $r_1$; indeed the equality $r_1$.B=$r_2$.B does not follow, because constraint $b_{V_{RS}}$ leaves the B attribute undetermined.*

*Still, the provenances of $R\ r_2$ and $R\ r_1$ are distinct ($v_r$, respectively $v_{rs}$), so the tpa-chase records the fact that the set $A$ of relational and equality atoms involving $r_1, s_1$ would have been created by the isolated standard chase of the $U$-subquery induced by $v_{rs}$ (as illustrated in Example 2.3, when chasing subquery $R_2$). The tpa-chase step is therefore allowed to add $A$ to $U_2$, adorning the tuple variables $r_1, s_1$ with provenance term $v_{rs}$ (see $U^{\mathcal{C}\mathcal{V}}$ in Example 3.2).*

*I4: disallow reapplication of a chase step with same constraint and same premise.* As seen in Example 3.4, case *I3* occurs when at least one of the relational atoms in the conclusion of $d$ 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 keep track of provenance as desired, its divergence from the standard chase step would immediately lead to non-termination because the same chase step now applies infinitely often. Indeed, in Example 3.4 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 applies 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 re-applications 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. Function symbols used in calls must be distinct across constraints (so that the chase step with a constraint is not mistaken for a re-application of a chase step with a distinct constraint). Function calls should intuitively take as arguments all tuple variables of the premise. However, it turns out that to distinguish between non-identical atoms it is sufficient to consider fewer function arguments: namely, those attributes of the premise tuples that also appear in (the equalities of) the conclusion.

EXAMPLE 3.5. *We illustrate only for constraint $b_{V_R}$, whose normalization involves setting the undetermined attribute $r$.B equal*

*to a function call:* $\forall v_r, v_r \in V_R \rightarrow \exists r, r \in R \wedge r.A = v_r.A \wedge r.C = v_r.C \wedge r.B = f(v_r.A,\ v_r.C)$

Interestingly, these functions correspond to the classical *Skolem functions* one would obtain when eliminating existential quantifiers from the constraints if written in First-Order Logic form. Notice that the attributes that were undetermined in the original form of the dependencies are now determined by the Skolem terms, in short *Skolem-determined*.

**Skolem chase.** We call the (provenance-unaware) chase flavor using these Skolem functions the *Skolem chase*. When checking whether an atom with a Skolem-determined attribute is "already there", the Skolem chase step requires an identical copy thereof in $q$ (Skolem function calls only match calls with the same function symbol and arguments). We can show that the Skolem chase is essentially equivalent to the standard chase in terms of its termination behavior and its result. Denoting with $Sk(\mathcal{C})$ the skolemized version of set $\mathcal{C}$ of dependencies, we have:

THEOREM 3.2. *(i) If $\mathcal{C}$ is a weakly-acyclic set of dependencies, then the Skolem chase with $Sk(\mathcal{C})$ terminates. (ii) For every SPJR query $q$, the result of the standard chase of $q$ with $\mathcal{C}$ is equivalent to the result of the Skolem chase of $q$ with $Sk(\mathcal{C})$.*

We can further characterize the comparison between the two chases in terms of their respective number of steps. In the weakly-acyclic case, [15] shows an upper bound on the length of every standard chase sequence. This bound is polynomial in the size of the instance when fixing its schema. While in some cases, as shown in the example, the Skolem chase sequences may be longer than the standard chase ones, we can prove that for weakly-acyclic dependencies the upper bound exhibited in [15] also applies to the Skolem chase.

**pa-chase.** We now revisit cases *I1* through *I3*, which prescribe the behavior of the tentative tpa-chase step. This was initially modeled on the standard chase step, leading to the non-termination problem described in case *I4*. We adjust this design by making the pa-chase record the provenance of atoms constructed by the Skolem 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 Skolem chase step with the skolemized version of $d$, denoted $Sk(d)$. We can now formally prove that, defined in this way, the pa-chase maintains invariant ($\diamond$):

THEOREM 3.3. *The provenance of an atom $a$ constructed during the pa-chase of the universal plan specifies precisely the set of universal plan subqueries whose Skolem chases (conducted in isolation from each other) would construct $a$.*

We can show that Invariant ($\diamond$) suffices for our purposes, since Theorems 3.3 and 3.2 imply the completeness of the $Prov_{C\&B}$ algorithm (Theorem 3.1).

# 4. IMPLEMENTATION

We present some of the key techniques we employed in implementing the $Prov_{C\&B}$ reformulation algorithm.

**Standard chase** Our chase implementation incorporates several optimizations that are summarized below:

*Chase step as query evaluation.* A chase step searches for homomorphic matches of the premise and conclusion of constraint $d$ against the query $Q$. The search for homomorphic matches of the premise can be modeled as running $d$'s premise $d_P$ (viewed as a query) against $Q$ (viewed as a symbolic database known in the literature as the canonical instance of $Q$ [1]). For instance, for constraint $b_{V_{RS}}$ in Example 2.1, matches of the premise are found by the natural join of tables $R$ and $S$. Extensions of these matches to the conclusion $d_C$ of $d$ are modeled analogously. We implement the search for matches as query evaluation. We compile $d_P$ to a query plan based on relational algebra operators, and we run it over an internal representation of $Q$ using its canonical instance.

*Standard query optimizations.* Modeling 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 introducing 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 [29].

**Pa-chase** Our provenance-aware chase implementation reuses the core design choices listed above for the standard chase, adapted of course to account for the Skolem chase and provenance bookkeeping. As a side-effect of our reformulation work, the pa-chase implementation delivers a minimal-why-provenance-tracking processor for conjunctive queries (generalized to support invention of values using Skolem functions).

# 5. EXPERIMENTS

We evaluated our $Prov_{C\&B}$ implementation in a recreation (and extension) of the setting described in [29] 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 ([29])** 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 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 $r_2.A_2 = s_{22}.A_2$

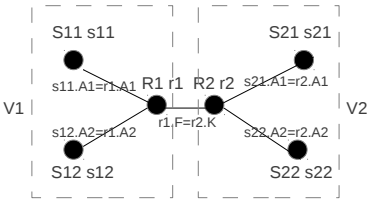


*Figure 1: Join graph for a two-star query*

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 2.

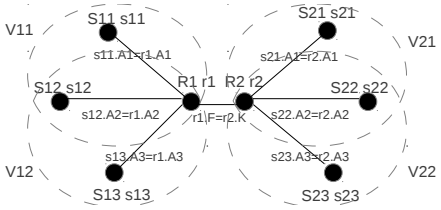


*Figure 2: Chain-of stars configuration with 3 corners*

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 [21].

**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[5].

Figure 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

[5]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).

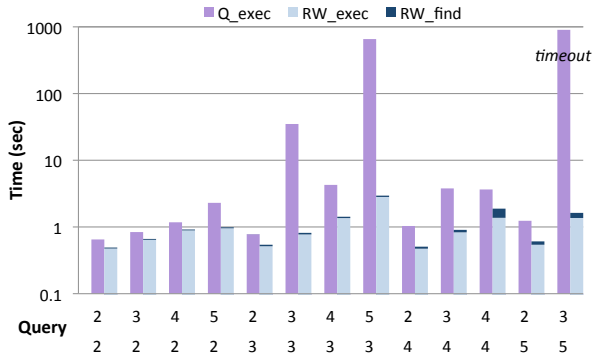


*Figure 3: Elapsed times on one database instance*

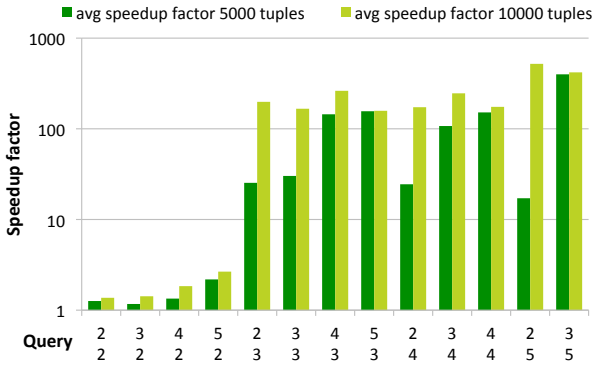


*Figure 4: Average speedup factors on 10 database instances*

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 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 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 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 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.

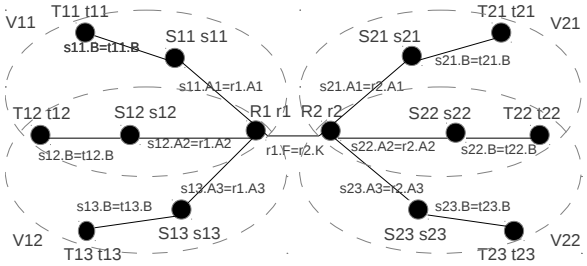


*Figure 5: Extended chain-of-stars configuration*

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. We omit their values, focusing instead on reporting $RW_{find}$ in Figure 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 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*

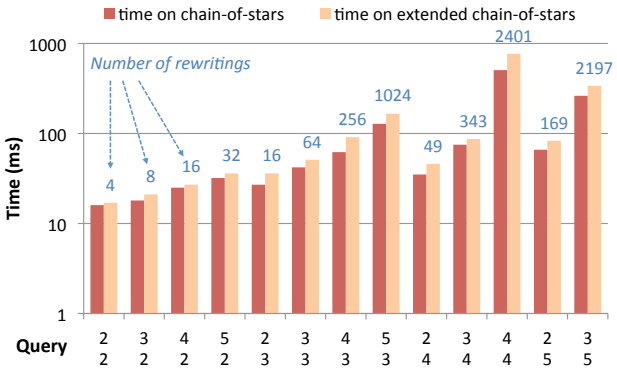


*Figure 6: Rewrite computation times ($RW_{find}$)*

backchase (called the Full Backchase in [29]) 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 6 as bar labels. We note the exponential trend of savings as the number of hubs or corners increases.

# 6. 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 [22, 3, 2, 8, 31], to going from set semantics to bag or mixed bag-set semantics (see [9] and the references therein), to adding limited access patterns for the views [16, 26], or to using potentially infinite sets of views [23]. In the context of information integration, view-based rewriting has been extended also to finding not-necessarily equivalent (but maximally-contained) rewritings (see [19] 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 [22], 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., [17, 20, 33, 5]), 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*, discussed at length in this paper, is the only complete algorithm we are aware of in this setting.

Our techniques rely on the classic chase procedure [1]. Recently, we have witnessed revived interest in the chase, with studies such as [24] focusing in particular on more permissive conditions that can guarantee termination.

In the pa-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 [6], and corresponding to a particular case of a provenance semiring [18].

# 7. DISCUSSION

**Chase termination beyond weak acyclicity.** We have proven that the pa-chase terminates on weakly-acyclic sets of constraints, as does the standard chase. However, the latter is guaranteed to terminate for a host of more permissive conditions (e.g. see [24]). We

are optimistic that the pa-chase terminates for these classes of constraints as well, thus extending the $Prov_{C\&B}$ completeness result to them. We leave this investigation to future work.

**Selecting among minimal reformulations.** Our experiments have shown that the $Prov_{C\&B}$ can enhance an optimizer's performance even when run on top of the DBMS, using a lightweight cost model to select a rewriting (the number of joins in this case). The advantage of such an approach is to avoid integration within the DBMS (a desirable, but logistically challenging goal). Such integration is avoidable even while using the DBMS's own cost estimator, typically accessible via an API. [29] sketches (and [28] details) a backchase which grows $U$-subqueries bottom-up from smallest to largest and prunes subqueries as soon as their cost exceeds that of the best reformulation found so far. This strategy returns a cost-optimum reformulation for a large class of cost models, called "monotonic" because they estimate the cost of a query to be at least the cost of its subqueries. This idea carries over to the $Prov_{C\&B}$, only much more efficiently, since the bottom-up cost-based pruning would cost-estimate only the subqueries of the already enumerated minimal reformulations (as opposed to all $U$-subqueries), memoizing the cost of common subqueries. It turns out that one can do even better by interleaving the enumeration of minimal rewritings with their costing into a cost-based-pruning pa-chase guaranteed to still find all cost-minimum rewritings (the detailed development is beyond the scope of this paper).

**Particularly well-suited $Prov_{C\&B}$ applications.** Our experiments evaluate the $Prov_{C\&B}$ in a traditional query processing setting. There, partial view-based rewritings are of interest (these are allowed to mention base tables besides the views). Clearly there are exponentially more partial rewritings than total ones (in which only views can be mentioned), so the $Prov_{C\&B}$ can find them even faster (all the more so since often there is no total rewriting: see our own experiments for instance, where the hub tables had to be part of the rewriting). Scenarios that care only for total rewritings and require the overall optimum among them are therefore the ideal $Prov_{C\&B}$ applications (query processing does not fit this bill, even though the $Prov_{C\&B}$ is useful there as we have seen).

One such scenario is access control enforcement via *security views* [25, 30]. Here, a query is considered "safe" only if it has a total rewriting using a set of legal views. In previous work, the existence of such rewriting sufficed for the query to be safe. 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 she can go request it. This involves selecting the minimum-clearance total rewriting(s).

**What about NP-hardness?** One may wonder how the fast reformulation times we measured square with the NP-hardness of the view-based rewriting problem even in the absence of constraints [22]. The answer is that there is another classical NP-hardness result, namely for evaluating SPJR queries against a database, which commercial DBMSs haven taken in stride since their inception, even for large databases, as it refers to query size only. Our contribution is to reduce reformulation to query evaluation (over toy-sized symbolic databases), opening up the bag of tools of relational query processing. The analogy with query evaluation had been applied in prior *C&B* implementations to speed up standard chase step evaluation and containment mapping search, but it was not clear how the backchase phase could profit from it. The novel pa-chase establishes just such an analogy between the search among universal plan subqueries and (provenance-tracking) query evaluation.

# 8. REFERENCES


[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] 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.

[3] F. N. Afrati, C. Li, and P. Mitra. Answering queries using views with arithmetic comparisons. In *PODS*, 2002.

[4] C. Beeri and M. Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.

[5] 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.

[6] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[7] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[8] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM TODS*, 31(2), 2006.

[9] D. DeHaan. Equivalence of nested queries with mixed semantics. In *PODS*, pages 207–216, 2009.

[10] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, 1999.

[11] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[12] A. Deutsch and V. Tannen. Mars: A system for publishing xml from mixed and redundant storage. In *VLDB*, pages 201–212, 2003.

[13] A. Deutsch and V. Tannen. Reformulation of XML queries and constraints. In *ICDT*, 2003.

[14] R. Fagin. Horn clauses and database dependencies. *J. ACM*, 1982.

[15] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[16] 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.

[17] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, 2001.

[18] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[19] A. Halevy. Answering queries using views: A survey. *VLDB J.*, 2001.

[20] 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.

[21] M. Levene and G. Loizou. Why is the snowflake schema a good data warehouse design? *Information Systems*, 28(3):225 – 240, 2003.

[22] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[23] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external query processors. *J. Comput. Syst. Sci.*, 1999.

[24] M. Meier, M. Schmidt, and G. Lausen. On chase termination beyond stratification. *PVLDB*, 2(1):970–981, 2009.

[25] A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *ICDE*, 1989.

[26] A. Nash and B. Ludäscher. Processing first-order queries under limited access patterns. In *PODS*, 2004.

[27] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested xml queries using nested views. In *SIGMOD*, 2006.

[28] L. Popa. *Object/relational Query Optimization with Chase and Backchase*. PhD thesis, University of Pennsylvania, 2000.

[29] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far? In *SIGMOD*, pages 273–284, 2000.

[30] 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.

[31] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *VLDB*, 1996.

[32] C. Yu and L. Popa. Constraint-based xml query rewriting for data integration. In *SIGMOD*, pages 371–382, 2004.

[33] 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.