Optimizing the Chase: Scalable Data Integration under Constraints

George Konstantinidis
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292
konstant@usc.edu

José Luis Ambite
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292
ambite@isi.edu

ABSTRACT

We are interested in scalable data integration and data exchange under constraints/dependencies. In data exchange the problem is how to materialize a target database instance, satisfying the source-to-target and target dependencies, that provides the certain answers. In data integration, the problem is how to rewrite a query over the target schema into a query over the source schemas that provides the certain answers. In both these problems we make use of the chase algorithm, the main tool to reason with dependencies. Our first contribution is to introduce the *frugal* chase, which produces smaller universal solutions than the standard chase, still remaining polynomial in data complexity. Our second contribution is to use the frugal chase to scale up query answering using views under LAV weakly acyclic target constraints, a useful language capturing RDF/S. The latter problem can be reduced to query rewriting using views without constraints by chasing the source-to-target mappings with the target constraints. We construct a compact graph-based representation of the mappings and the constraints and develop an efficient algorithm to run the frugal chase on this representation. We show experimentally that our approach scales to large problems, speeding up the compilation of the dependencies into the mappings by close to 2 and 3 orders of magnitude, compared to the standard and the core chase, respectively. Compared to the standard chase, we improve online query rewriting time by a factor of 3, while producing equivalent, but smaller, rewritings of the original query.

1. INTRODUCTION

We witness an explosion of available data in all areas of human activity, from large scientific experiments, to medical data, to distributed sensors, to social media. Integrating data from disparate sources can lead to novel insights across scientific, industrial, and governmental domains. This integration is achieved by either creating a data warehouse, that is, by copying/transforming the data to a centralized site under a single schema for subsequent analysis (*data exchange* [14]), or by leaving the data at their original sources and querying the data at analysis time (*(virtual) data integration* [20]).

Formally, a *data integration/exchange setting*, $< S, T, \Sigma_{st}, \Sigma_t >$, is a tuple where S is a set of heterogeneous schemas from multiple sources, T is the target schema (aka the domain, global, or mediated schema), which must satisfy a set of target constraints/-dependencies Σ_t, and Σ_{st} is a set of schema mappings that transform the data in schemas S into schema T. The main problem in this setting, and the focus of this paper, is conjunctive query answering (aka *answering queries using views under dependencies* [19], or *computing certain answers under constraints* [14, 1, 11]): the user poses a query over target schema T and the system needs to provide the *certain* answers to the query using the data from the sources S such that all constraints Σ_{st} and Σ_t are satisfied. Data exchange and virtual integration are the two main approaches for getting the certain answers. Data exchange achieves this by materializing a target database, by obtaining data from the sources using (satisfying) the schema mappings Σ_{st}, and satisfying the target constraints Σ_t. Then, the certain answers to a user query can be computed by directly evaluating the query over the materialized target database. In virtual integration, a mediator uses the schema mappings and target constraints to rewrite the target user query into a query that only uses terms from the source schemas but produces all the certain answers.

Mappings in Σ_{st}, and target constraints in Σ_t, are usually given in the form of logical implications of conjunctive formulas, called *tuple-generating dependencies* (TGDs) [6] or Global-Local-as-View (GLAV) rules [15]. TGDs are very expressive, but query answering (and rewriting) under general target TGDs is undecidable [6]. However, some syntactic restrictions, such as *weakly-acyclic TGDs* [14], are decidable (even data-complexity tractable) in query answering.

In data exchange, the main technique to reason with constraints is *the chase* [6], a form of forward chaining that "fetches" data from the sources through the mappings to a target database, and also "completes" this database w.r.t. the target constraints. One can then disregard the constraints and do query answering over the completed database. In this paper, we present an optimization of the chase algorithm usable in data exchange with GLAV mappings and standard-chase terminating (e.g., weakly-acyclic) target TGDs.

In virtual integration (which is achieved through query rewriting), the initial focus has been on settings without target constraints, i.e., $\Sigma_t = \emptyset$, and mappings with only one source predicate in the rule antecedent, called Local-as-view (LAV) mappings [27, 12]. LAV mappings expose the core challenges in query rewriting, since they contain joins of existential variables. Query rewriting with LAV mappings and no constraints, aka *answering queries using views*, has been extensively studied in query optimization, data integration and other areas [20, 26]. In previous work, we developed a scalable query rewriting algorithm for LAV mappings and no target constraints, named GQR [24], that is about two orders of magnitude faster than the previous state-of-the-art (Minicon [31], MCDSAT [5]), and rewrites a conjunctive query in the presence of

.

10,000 LAV mappings in under 1 second. For virtual integration with target constraints, Afrati and Kiourtis [1] used the chase algorithm in a novel way by "compiling" the target constraints (specifically, LAV weakly-acyclic TGDs) into the LAV mappings to reduce the problem to view-based query rewriting without constraints. In this paper, we present an algorithm for query rewriting under LAV weakly-acyclic target TGDs, building on [1, 24], and our optimized chase. This type of constraints is first-order rewritable, includes practically interesting languages like RDF/S, and has good computational properties in data integration, data exchange [16, 1], and in inconsistent and incomplete databases [2].

In particular, our paper presents two main contributions:

1. The frugal chase. We develop a novel, optimized version of the standard chase, usable in data integration, data exchange, or incomplete database settings, with GLAV mappings and GLAV target constraints. Instead of adding the entire consequent to a solution when a chase rule is applicable, as in the standard chase, the frugal chase avoids adding provably redundant atoms (Sect. 3). We prove that the frugal chase results in equivalent, yet smaller in size (number of tuples), universal solutions with the same data complexity as the standard chase. We also present a procedural version of the frugal chase (Sect. 4), and a compact version of the latter adapted to our GQR rewriting algorithm (Sect.5).

2. A scalable conjunctive query rewriting algorithm for LAV mappings under weakly-acyclic LAV target constraints. This algorithm uses the compact frugal chase to *efficiently* compile the constraints into the mappings (à la [1]), and then efficiently do query rewriting (using [24]). Our compact frugal chase identifies common patterns across the mappings and "factorizes" them using a graph representation. This allows us to trigger a constraint for multiple views at the same time, and to add consequent predicates across multiple views at once. Identifying and indexing common patterns and chasing the mappings can be performed offline in a precompilation phase, independently of user queries, thereby speeding up system's online query performance. Our compact graph representation of the mappings is particularly tailored for GQR, optimizing our solution even more. Our algorithm experimentally performs about 2 orders of magnitude faster than running the standard chase on each view individually and then applying query rewriting using views. Our approach scales to larger numbers of constraints and views and produces smaller, but equivalent, UCQ rewritings (containing less and shorter conjunctive queries), that compute the certain answers. For our experimental setting the size of the frugal chased mappings is very close to the core [13, 10] (i.e., the globally minimized chased mappings). Nevertheless, our compact algorithm achieves this output almost 3 orders of magnitude faster than the core chase [10], since we do not rely on minimization.

2. BACKGROUND

Queries. We use rule notation for conjunctive queries. The following query asks for doctors responsible for discharging patients:

q(d)← Doctor(d), DischargesPatientFromClinic(d,p,c)

The antecedent of the query is called the *body* of the query while $q(d)$, the consequent, is the *head*. Variables in the head are called *distinguished* (e.g., d); the rest are *existential* variables (e.g., p, c). A union of conjunctive queries (UCQ) is a set of same-head rules. We denote $vars(S)$ (or $cons(S)$) the set of variables (or constants) of a query or a set of predicates S.

Query Containment. Query Q_1 is contained in query Q_2, $Q_1 \subseteq Q_2$, iff for all databases D the result of evaluating Q_1 on D, $Q_1(D)$, is contained in the result of evaluating Q_2, that is, $Q_1(D) \subseteq Q_2(D)$.

Homomorphism. Given two sets of atoms S_1, S_2, a homomorphism from S_1 to S_2 is a function $h{:}vars(S_1) \cup cons(S_1) \rightarrow vars(S_2) \cup cons(S_2)$, such that: (1) h is the identity on constants, and (2) for all atoms $A \in S_1$, $h(A) \in S_2$ (a homomorphism h is extended over atoms, sets of atoms, and queries in the obvious manner). Given two conjunctive queries Q_1, Q_2, a **containment mapping** from Q_2 to Q_1 is a homomorphism $h{:}body(Q_2) \rightarrow body(Q_1)$ s.t. $h(head(Q_2))$ = $head(Q_1)$. It holds that $Q_1 \subseteq Q_2$ iff there is a containment mapping from Q_2 to Q_1 [9].

Dependencies are used to define schema mappings and target schema constraints. Tuple-generating dependencies (or GLAV rules), are formulas of the form: $\forall \vec{x}, \vec{y}\ \phi_A(\vec{x}, \vec{y}) \rightarrow \exists \vec{z}\ \psi_B(\vec{x}, \vec{z})$, with ϕ_A and ψ_B conjunctive formulas over sets of predicates A and B, and $\vec{x}, \vec{y}, \vec{z}$ tuples of constants or variables. A TGD with a single predicate in the consequent is called a Global-as-View (GAV) rule. A TGD with a single predicate in the antecedent is called a Local-as-View (LAV) rule [20]. When A and B are the sets of source and target predicates, respectively, the formulas are called *source-to-target tuple-generating dependencies* (st-TGDs) [14]. When $A = B$ is the set of target predicates, the formulas are called target TGDs. A second important class of constraints (not considered in this paper) involves equality-generating dependencies (EGDs), which are generalizations of functional dependencies.

Containment under Dependencies. Query Q_1 is contained in query Q_2 under a set of constraints Σ, denoted $Q_1 \subseteq_\Sigma Q_2$, iff for all databases D that are consistent with Σ, $Q_1(D) \subseteq Q_2(D)$

Weakly Acyclic TGDs (wa-TGDs) [14]. Let Σ be a set of TGDs over schema $\Re = \{R_1, ..., R_n\}$. Construct a directed graph, called the dependency graph, as follows: (1) there is a node for every pair (R_i, A) with A an attribute of R_i, call such pair (R_i, A) a position; (2) add edges as follows: for every TGD $\forall \vec{x}, \vec{y}\ \phi(\vec{x}, \vec{y}) \rightarrow \exists \vec{z}\ \psi(\vec{x}, \vec{z})$ in Σ, for every x in $\vec{x}$ that occurs in ψ and for every occurrence of x in ϕ in position (R_i, A_i): (2.1) for every occurrence of x in ψ in position (R_j, B_k), add an edge $(R_i, A_i) \rightarrow (R_j, B_k)$ (if it does not already exist), and (2.2) for every existentially quantified variable $z \in \vec{z}$ and for every occurrence of z in ψ in position (R_t, C_m), add a special edge $(R_i, A_i) \rightsquigarrow (R_t, C_m)$ (if it does not already exist). Then Σ is weakly acyclic (wa) if the dependency graph has no cycle with a special edge. **LAV Weakly Acyclic TGDs** are wa-TGDs that have a single predicate in the antecedent.

The Chase is useful to reason with dependencies. Given a conjunction of atoms B (we will run the chase on both database instances and on consequents of mappings) and a TGD $\sigma = \forall \vec{x}, \vec{y}\ \phi(\vec{x}, \vec{y}) \rightarrow \exists \vec{z} \psi(\vec{x}, \vec{z})$, then σ is *applicable* to B with *antecedent homomorphism* h iff h is a homomorphism from ϕ to B (intuitively, the antecedent holds in B), such that h cannot be extended to cover ψ (i.e., the consequent is not already satisfied). We *apply* the TGD σ by adding its consequent to B. Formally, a *chase step* adds $\psi(h(\vec{x}), f(\vec{z}))$ to B, where h is the antecedent homomorphism and f creates "fresh" variables, known as *skolems* or *labeled nulls*, for all the existential variables ($\vec{z}$). The *standard chase* is an exhaustive series of chase steps, and may be finite or infinite depending on the constraints. We denote $chase_\Sigma(B)$ the result of chasing B with all constraints in Σ. Variations of the chase are the *parallel* and *core* chase [10]. Every parallel chase step decides all applicable constraints on B first, and then adds to B their consequents. Each core chase step is a parallel step followed by minimization of the result.

Data Exchange. Given a finite source database instance I, a set of st-TGDs Σ_{st}, and a set of target dependencies Σ_t, the *data exchange problem* [14] is to find a finite target database instance J, called a *solution*, such that I, J satisfy Σ_{st} and J satisfies Σ_t. The *certain answers* of a query q on the target schema obtained using the source instance I, denoted by $certain(q, I)$, is the set of all tuples t of constants from I such that for every solution J, $t \in q(J)$ [14]. For certain classes of TGDs we can reach a representative solution for the entire space of solutions, called a **universal solution**, which has homomorphisms to all other solutions, and $certain(q, I)$ can be computed by issuing q on it [14]. The chase, $chase_{\Sigma}(I)$, is a sound algorithm for finding universal solutions. The chase with general target TGDs might not terminate, so relevant research has focused in identifying chase-terminating classes of TGDs, such as wa-TGDs, for which the chase is both a sound and a complete algorithm for computing universal solutions.

(Virtual) Data Integration (VDI) is the problem of computing the certain answers of a target query by rewriting it as a query over the source schemas and querying the sources directly. Given a query Q, a set of GLAV schema mappings $\mathcal{M} = \{M_1, ..., M_n\}$, and no target constraints, Q' is a **maximally-contained rewriting [27]** of Q using $\mathcal{M}$ if: (1) Q' only contains source predicates, (2) $Q' \subseteq Q$, and (3) there is no query Q'' only containing source predicates, such that $Q' \subseteq Q'' \subseteq Q$ and $Q'' \not\equiv Q'$. When considering target constraints, this definition involves containment under dependencies ($\subseteq_{\Sigma}$). In this paper we reduce the query rewriting with constraints problem to query rewriting without target constraints. Our maximally-contained rewritings will compute the certain answers.

Data Integration with weakly acyclic LAV TGDs. We are interested in first-order rewritings (i.e., expressible in SQL) since we want to develop a practical solution that leverages scalable relational technology. When considering target constraints in data integration, maximally-contained first-order rewritings do not always exist. We focus on weakly-acyclic LAV constraints, which are first-order rewritable [16, 1]. We discuss other cases of first-order rewritable constraints in section 7. LAV wa-TGDs is a superset of the useful class of weakly-acyclic inclusion dependencies, as well as the class of LAV TGDs with no existential variables (LAV full TGDs), which can express important web ontology languages such as RDF/S. A maximally-contained UCQ rewriting under LAV wa-TGDs, which computes the certain answers, can be obtained by first chasing the views with the target constraints, and then applying a query answering using views algorithm (without constraints) [1].

THEOREM 1. ***Chasing the Views [1]*** *Given a query Q on a schema R, a set of LAV schema mappings $\mathcal{V} = \{V_1, ..., V_n\}$ on R, a source instance I under $\mathcal{V}$ and a set of weakly-acyclic LAV TGD constraints Σ on R, the set of certain answers $certain(Q, I)$ is computed by the UCQ maximally-contained rewriting of Q using $\{V_1{}', ..., V_n{}'\}$, where each $V_i{}' \in \{V_1{}', ..., V_n{}'\}$ is produced by running the standard chase on the consequent of V_i using Σ.*

Motivating Example. Consider the following LAV rules describing sources S_1-S_4 of medical data. S_1 contains physicians that treat patients with a chronic disease. S_2 records the physician responsible for discharging a patient from a clinic. S_3 is the same to S_1 but physicians are typed as Doctors. S_4 provides Surgeons.

S_1(d, s) $\rightarrow$ TreatsPatient(d, p), HasChronicDisease(p,s)
S_2(d, p, c) $\rightarrow$ DischargesPatientFromClinic(d, p, c)
S_3(d,s) $\rightarrow$ TreatsPatient(d,p), HasChronicDisease(p,s), Doctor(d)
S_4(d) $\rightarrow$ Surgeon(d)

The maximally contained (with no constraints) rewriting of q (presented in the beginning of this section) is: q'(d)$\leftarrow$$S_3$(d,s),$S_2$(d,z,c).

Now, consider the following (RDF/S) constraints that capture "domain" and "range" properties, and "subclass" relations. Constraint c_1 states that the domain of TreatsPatient is Doctor and the range is Patient. Constraint c_2 states that Surgeons are Doctors (as for queries our notation for constraints and views omits quantifiers).

c_1: TreatsPatient(x,y) $\rightarrow$ Doctor(x), Patient(y)
c_2: Surgeon(x) $\rightarrow$ Doctor(x)

Theorem 1 guarantees that we can answer query q, using sources S_1-S_4, by first chasing the consequents of the views and then looking for maximally-contained rewritings of the query using the chased views. Running the chase on S_1-S_4 using c_1 and c_2 yields:

S_1'(d,s)$\rightarrow$ TreatsPatient(d,p),HasChronicDisease(p,s), Doctor(d), Patient(p)
S_2'(d,p,c)$\rightarrow$ DischargesPatientFromClinic(d,p,c)
S_3'(d,s) $\rightarrow$ TreatsPatient(d,p),HasChronicDisease(p,s), Doctor(d), Patient(p)
$S_4{}'$(d) $\rightarrow$ Surgeon(d), Doctor(d)

The maximally-contained rewriting of q using $S_1{}'$-$S_4{}'$ is the UCQ: q'(d) $\leftarrow$ (S_1(d,x),S_2(d,y,z)) $\vee$ (S_3(d,u),S_2(d,v,w)) $\vee$ (S_4(d),S_2(d,s,t)). This approach was employed in [1] by running the standard chase on the views and using the Minicon algorithm [31] for query rewriting. In this paper, we chase the views using our optimized frugal chase (Sect. 3). Moreover, we develop a graph-based compact version of the frugal chase (Sect. 5) optimized for running on multiple views simultaneously, tailored to be input directly into GQR [24] for fast query rewriting. Running the frugal chase, rather than the standard one produces shorter (but equivalent) mappings (in number of predicates/joins), which in turn produce less and shorter (but equivalent) conjunctive queries in the final UCQ rewriting.

As an example of our approach, consider the mapping S_3 and constraint c_3 below, which states that an individual with a chronic disease must be a patient treated by a doctor:

c_3: HasChronicDisease(pat,dis) $\rightarrow$ TreatsPatient(doc,pat), Doctor(doc), Patient(pat)

Since there is no homomorphism that maps the consequent of the rule to the consequent of the view, the standard chase produces:

$S_3{}''$(d,s) $\rightarrow$ TreatsPatient(d,p), HasChronicDisease(p,s), Doctor(d), TreatsPatient(d2,p), Doctor(d2), Patient(p)

Our algorithm produces a shorter, yet equivalent, mapping:

$S_3{}'''$(d,s)$\rightarrow$TreatsPatient(d,p),HasChronicDisease(p,s),Doctor(d),Patient(p)

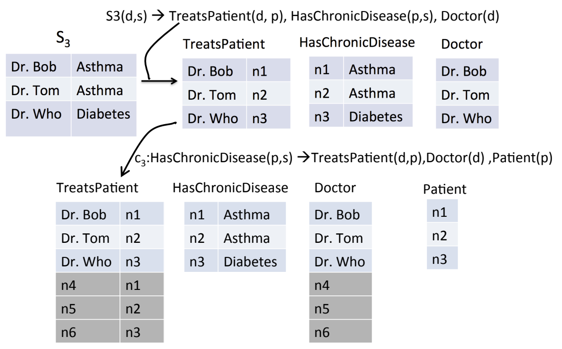
Consider query q_2(s)$\leftarrow$ TreatsPatient(d,p), HasChronicDisease(p,s). Using $S_3{}''$ this query will give the UCQ rewriting: q_2'(s) $\leftarrow$ S_3(d,s) $\vee$ S_3(d2,s). One of the two elements of q_2' is redundant. Minimizing the output of a query rewriting algorithm is an orthogonal NP-hard problem [27]. Query rewriting using the frugal chased mapping $S_3{}'''$ avoids this redundancy, without running minimization, leading to the smaller (and faster to evaluate) rewriting: q_2''(s) $\leftarrow$ S_3(d,s).

3. OPTIMIZING THE CHASE

Often, in the presence of existential variables/labeled nulls, some of the consequences of a chase rule are already implied in the database. That is, although the rule is not satisfied (i.e., there is no homomorphism from the consequent to the database instance that is an extension of the antecedent homomorphism), the consequent might be *partially* satisfiable; adding a subset of the consequent's atoms can construct such a homomorphism from the entire consequent. In this section we introduce the *frugal* chase, which is equivalent to the standard chase, but results in smaller universal solutions.

Consider the simple data exchange scenario of Fig.1 with a source described by S_3 and a target constraint c_3 (cf. Sect. 2). Existential variables in the constraint introduce additional labeled null tuples which are redundant (nulls do not participate in the certain answers of a query). The bottom 3 rows of TreatsPatient and Doctor can be removed. Our frugal chase avoids adding such redundant facts.

Before we define the frugal chase, let us introduce some useful notions. Let B a database instance (or any other conjunction of atoms such as the consequent of a TGD). The *Gaifman graph of nulls* [13] is an undirected graph with nodes being the existential variables (labeled nulls) of B; edges between nodes exist if the variables co-exist in an atom in B. Dually, *the Gaifman graph of facts* of B, denoted $grf(B)$, is the graph whose nodes are the atoms of B and two nodes are connected if they share existential variables/labeled nulls. Note that parts of B that are connected only through constants (as well as distinguished variables for TGDs), constitute different connected components of the Gaifman graph of facts. For G_i a connected component of $grf(B)$, $V(G_i)$ is the


S3(d,s) → TreatsPatient(d, p), HasChronicDisease(p,s), Doctor(d)
S3
TreatsPatient
HasChronicDisease
Doctor
c3:HasChronicDisease(p,s) → TreatsPatient(d,p),Doctor(d) ,Patient(p)
Patient

S3	
Dr. Bob	Asthma
Dr. Tom	Asthma
Dr. Who	Diabetes

TreatsPatient	
Dr. Bob	n1
Dr. Tom	n2
Dr. Who	n3

HasChronicDisease	
n1	Asthma
n2	Asthma
n3	Diabetes

Doctor
Dr. Bob
Dr. Tom
Dr. Who

TreatsPatient	
Dr. Bob	n1
Dr. Tom	n2
Dr. Who	n3
n4	n1
n5	n2
n6	n3

HasChronicDisease	
n1	Asthma
n2	Asthma
n3	Diabetes

Doctor
Dr. Bob
Dr. Tom
Dr. Who
n4
n5
n6

Patient
n1
n2
n3

Figure 1: Standard vs Frugal Chase.

set of all the facts (i.e., the nodes) in G_i. For any conjunction of atoms B we denote the *decomposition* of B to (the facts of) its connected components $\{G_i, ..., G_n\}$ as the set of sets (of facts) $dec(B) = \{V(G_i), ..., V(G_n)\}$. Fig. 2 shows in dotted circles the different connected components of TGD c and instance B.

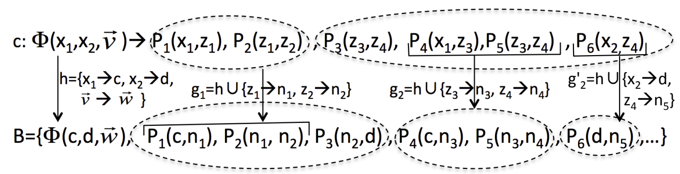
A constraint can be *decomposable* to an equivalent set of "simpler" constraints, each with a different element of its decomposition as consequent. For example, c_3 in Fig. 1 can be broken down to two constraints with consequents {TreatsPatient(d,p),Doctor(d)} and {Patient(p)} respectively. For Fig. 1, applying the standard chase using the new set of constraints would also avoid the redundancies presented. Nevertheless, our frugal chase produces smaller chase results, even with non-decomposable constraints.

Informally, a set of predicates is partially satisfiable, and not added during the frugal chase application, in two cases: if it is a connected component of the constraint and is mapped to the instance as a whole, or if its image on the database instance is a complete connected component of the instance. Any union of such partially satisfiable sets is a partially satisfiable set, as long as the individual satisfying mappings agree on their common arguments.

DEF. 1. ***Partially Satisfiable Set.*** *Let σ be a TGD constraint $\forall \vec{x}, \vec{y}, \phi(\vec{x}, \vec{y}) \rightarrow \exists\, \vec{z}\, \psi(\vec{x}, \vec{z})$ and B an instance s.t. there is an antecedent homomorphism h that maps $\phi(\vec{x}, \vec{y})$ to B. A set of atoms $S \subseteq \psi$, is partially satisfiable for h if there exists an extension of h, h', called a satisfying homomorphism for S, s.t. $h'(S) \subseteq B$ and for each $S_i \in dec(S)$, either:*

1. *(a) for all existential variables $\vec{z_i}$ in S_i, $h'(z_i)$ is an existential variable (or labeled null) in B, and*
   (b) $h'(S_i) \in dec(B)$, i.e., the image of S_i is an entire connected component of B; or
2. *$S_i \in dec(\psi)$, i.e., the mapped set S_i is actually an entire connected component of the constraint consequent.*

We illustrate Def. 1 with the example in Fig. 2, which shows how the frugal chase applies constraint c on instance B (Φ is a conjunction of atoms, c and d are constants, x_i and z_i are variables, n_i are labeled nulls, and P_i are relations). Consider whether the set of predicates $S = \{P_1(x_1, z_1), P_2(z_1, z_2), P_4(x_1, z_3), P_5(z_3, z_4)\}$, a subset of the consequent of constraint c, is partially satisfiable. The decomposition $dec(S)$ has two elements (connected components): $S_1 = \{P_1(x_1, z_1), P_2(z_1, z_2)\}$ and $S_2 = \{P_4(x_1, z_3), P_5(z_3, z_4)\}$. Per Def. 1, for our S there exists a homomorphism h' which extends the one from the antecedent (h), such that $h'(S) \subseteq B$. Moreover, S_1 is an entire connected component of the constraint (falling in case 2 of Def. 1), while for S_2 all its existential variables are mapped through h' (and in particular g$_2$) to existential variables (case Def. 1.1(a)), and $h'(S_2)$ is a connected component of B (case Def. 1.1(b)). So S is partially satisfiable. Note that $P_6(x_2, z_4)$ in the constraint could be partially satisfiable by mapping it to


c: $\Phi(x_1,x_2,\vec{v}) \rightarrow P_1(x_1,z_1), P_2(z_1,z_2), P_3(z_3,z_4), P_4(x_1,z_3), P_5(z_3,z_4), P_6(x_2,z_4)$
$h=\{x_1 \rightarrow c, x_2 \rightarrow d, \vec{v} \rightarrow \vec{w}\}$
$g_1 = h \cup \{z_1 \rightarrow n_1, z_2 \rightarrow n_2\}$
$g_2 = h \cup \{z_3 \rightarrow n_3, z_4 \rightarrow n_4\}$
$g'_2 = h \cup \{x_2 \rightarrow d, z_4 \rightarrow n_5\}$
$B=\{\Phi(c,d,\vec{w}), P_1(c,n_1), P_2(n_1, n_2), P_3(n_2,d), P_4(c,n_3), P_5(n_3,n_4), P_6(d,n_5),...\}$

Satisfying homomorphism is either $h' = [g_1 \cup g_2]$ OR $h'' = [g_1 \cup g'_2 \cup f]$

frugal-chase(B,c) = $B \cup h'(\{P_3(z_3,z_4), P_6(x_2,z_4)\}) = B \cup \{P_3(n_3,n_4), P_6(d,n_4)\}$

$\equiv B \cup h''(\{P_3(z_3,z_4),P_4(x_1,z_3),P_5(z_3,z_4)\}) = B \cup \{P_3(n_6,n_5),P_4(c,n_6),P_5(n_6,n_5)\}$

Figure 2: Frugal Chase: Partially Satisfiable Set.

$P_6(d, n_5)$ in B; however not at the same time as $\{P_4(x_1, z_3), P_5(z_3, z_4)\}$, which maps z_4 to a different variable, n_5 (according to Def. 1 there needs to be a single satisfying homomorphism h' which maps all elements of a partially satisfiable set). Hence, the set of partially satisfiable predicates of a constraint might not be unique; in Fig. 2 there are two equivalent alternative frugal chases, given by h' and h''. Function f in h'' maps variables that are not in the domain of the satisfying homomorphism to fresh names. This non-uniqueness does not cause a problem for the end chase result; as we prove later, all frugal chase results are universal solutions. Currently our algorithms choose one arbitrary partially satisfiable set (our implementation chooses the first it discovers). Nevertheless, one could develop heuristics to materialize the preferred alternative, e.g., the shorter chase or one depending on the application.

The frugal chase is *applicable* whenever not all atoms of the consequent are in a partially satisfiable set; by Lemma 1 this is equivalent to the standard chase applicability (proof omitted due to space).

LEMMA 1. *Let σ a TGD constraint $\forall \vec{x}, \vec{y}, \phi(\vec{x}, \vec{y}) \rightarrow \exists \vec{z}\, \psi(\vec{x}, \vec{z})$ and B an instance. σ is applicable to B, per the standard chase, iff there exists an antecedent homomorphism h from $\phi(\vec{x}, \vec{y})$ to B and there exists one consequent predicate $P_\sigma(\vec{x_P}, \vec{z_P}) \in \psi(\vec{x}, \vec{z})$ (with $\vec{x_P} \subseteq \vec{x}$, $\vec{z_P} \subseteq \vec{z}$) which is not partially satisfiable for h.*

The *application* step of the frugal chase rule adds all such non partially-satisfiable atoms of the consequent to our database instance. Formally, for S a partially satisfied set of predicates of the consequent ψ with satisfying homomorphism h', $\overline{\psi}(\vec{x}, \vec{z}) = (\psi(\vec{x}, \vec{z}) \setminus S)$ is the set of all non partially satisfiable atoms w.r.t. S. The frugal chase step adds $\mathcal{H}(\overline{\psi}'(\vec{x}, \vec{z}))$ to our database, where the *applicable* homomorphism, $\mathcal{H}$, is an extension of h', mapping variables not appearing in h' to new "fresh" labeled nulls. In the example of Fig. 2, if we choose to satisfy $\{P_4(x_1, z_3), P_5(z_3, z_4)\}$, then $\mathcal{H} = h'$ and only the two non partially satisfiable atoms $P_3(n_3, n_4)$ and $P_6(d, n_5)$ are added to our database (compared to the entire consequent of c that the standard chase would add). The resulting instance after the frugal chase step is equivalent (but smaller in size) to the one produced by the standard chase.

THEOREM 2. *For all instances B, and sets of TGDs Σ, the frugal chase step is applicable to B with antecedent homomorphism h iff the standard chase step is applicable with h. Moreover if B' is the instance after the application of the standard chase step for h and $\sigma \in \Sigma$ and B'' is the instance after the application of the frugal chase step for h and σ, then B' and B'' are homomorphically equivalent, and they both satisfy σ (for h). [Proof in appendix].*

As in the standard chase, the *frugal chase* is an exhaustive series of frugal chase application steps. Since the output of each frugal chase step might be smaller, this might also lead to fewer number of chase steps overall (since redundant predicates triggering the application of subsequent constraints might not appear at all during the frugal chase). This optimizes our proposed solution even more.

THEOREM 3. *For all instances B, and sets of TGDs Σ, the frugal chase terminates for all instances and constraints for which the standard chase terminates, producing a universal solution.*

PROOF. Let $c_1, c_2, ...$ be an ordering of constraints (possibly repeating with different antecedent homomorphisms) in the frugal chase application. Let $fchase_{c_i}(B)$ and $chase_{c_i}(B)$ the results of the frugal and the standard chase resp. on B with c_i. $fchase_{c_1}(B)$ and $chase_{c_1}(B)$ are homomorphically equivalent, by Th. 2. Let g_1 the homomorphism from $chase_{c_1}(B)$ to $fchase_{c_1}(B)$. Let a constraint c_j satisfied, per the standard chase, on $chase_{c_1}(B)$, with satisfying homomorphism g_2. Composing g_2 with g_1, proves that the constraint is also satisfied on $fchase_{c_1}(B)$. Hence, $fchase_{c_1}(B)$ triggers at most as many constraints as $chase_{c_1}(B)$ does. Also, assuming c_2 is applicable to $fchase_{c_1}(B)$, it is also to $chase_{c_1}(B)$, with the same antecedent homomorphism. Moreover, after the application of c_2, $fchase_{c_2}(fchase_{c_1}(B))$ is homomorphically equivalent to $chase_{c_2}(chase_{c_1}(B))$ (proof is omitted due to space, but is very similar to our appendix proof). Hence, inductively, if the standard chase terminates on B with Σ so does the frugal chase with $fchase_{\Sigma}(B)$ satisfying all constraints and being a solution. In order to show that it is universal, i.e. has homomorphisms to all other solutions, we note that it has a homomorphism to $chase_{\Sigma}(B)$ which in turn has homomorphisms to all other solutions. □

The intuition behind our frugal chase is taking care that when adding a non-partially-satisfiable predicate, we will not introduce a join among this predicate and some relation, or some constant, in B which is not in the constraint (such a join would not be introduced by the standard chase). This case is avoided when satisfying an entire connected component of the constraint since there is nothing else (existentially) joining with that component in the consequent to be added (e.g., predicates P_1 and P_2 of c in Fig. 2). If our set of satisfied predicates (e.g., P_4 and P_5 of c in Fig. 2) is not an entire connected component of the consequent, its image has to be; otherwise whatever these predicates join with in the constraint (which we will add to our instance), will "accidentally" join with whatever joins with their image. For example, consider:

c_4: $P(x), R(x, z) \rightarrow P_1(x, y, z), P_2(y, w), P_4(y)$
$B = \{P(d), R(d, c)P_1(d, y_1, c), P_2(y_1, w_1), P_3(y_1)\}$

Running the standard chase with c_4 on B produces:

$B' = \{P(d), R(d, c)P_1(d, y_1, c), P_2(y_1, w_1), P_3(y_1),$
$P_1(d, y, z), P_2(y, w), P_4(y)\}$

Had we assumed partial satisfiability for both P_1, P_2 we would get:

$B''= \{P(d), R(d, c), P_1(d, y_1, c), P_2(y_1, w_1), P_3(y_1), P_4(y_1)\}$.

B'' and B' are not equivalent since there is no homomorphism from B'' to B', since the join of P_3-P_4 in B'' cannot be preserved. If P_3 was missing from B (we would fall in case 1: all joins of y_1 in B would be with images of partially satisfiable predicates) or if P_4 was missing from c_4 (we would fall in case 2: the entire connected component in the constraint is partially satisfiable), then we would end up in B' and B'' either both missing P_3 or both missing P_4 in which case they would be homomorphically equivalent.

Complexity. The problem of deciding whether the standard chase is applicable on an instance is polynomial in data complexity [14]. The difference to our case is essentially case 1(b) of Def. 1, which introduces a polynomial traversal of the corresponding connected component of the database, for all S_i. So, the frugal chase remains polynomial in data complexity. Notice that our definitions care for only one (arbitrarily chosen) partially satisfiable set. Potentially, we could exhaustively examine all subsets of a constraint's consequent; still polynomial in data complexity. Moreover, as our experiments attest for LAV constraints, checking partial satisfiability and running the frugal chase is faster than the standard in practice.

4. PROCEDURAL FRUGAL CHASE

In preparation to use the frugal chase in query rewriting using GQR, we present an alternative definition that examines each atom separately and decides whether it is partially satisfiable.

DEF. 2. *Let σ be a TGD constraint $\forall\vec{x}, \vec{y}, \phi(\vec{x}, \vec{y}) \rightarrow \exists\vec{z}\,\psi(\vec{x}, \vec{z})$, and B an instance s.t. there exists a homomorphism h: $\phi(\vec{x}, \vec{y}) \rightarrow B$. For all atoms $P(\vec{x_P}, \vec{z_P}) \in \psi(\vec{x}, \vec{z})$, $P(\vec{x_P}, \vec{z_P})$ is partially satisfiable for h if there exists an extension of h, h', called a satisfying homomorphism for P, s.t. $h'(P) \subseteq B$ and for all $z \in \vec{z_P}$:*

1. *if $h'(z)$ is an existential variable (labeled null for instances) then for every atom R_B in B that contains $h'(z)$, there is an atom R_C in the constraint that contain z, in the same argument positions, s.t.: (a) R_C is partially satisfiable for h' (its satisfying homomorphism is an extension of h'), (b) R_B is the image of R_C through the satisfying homomorphism for R_C, and (c) for all $R(\vec{x}, \vec{z}) \in \psi(\vec{x}, \vec{z})$ that contains z, if R is partially satisfiable for h, it is also partially satisfiable for h' (which, recursively, means it uses an extension of h');*
2. *if $h'(z)$ is (a) a constant (for instances or mappings), or (b) a distinguished variable (for mappings), or (c) an existential variable (labeled null for instances) which does not fall into case (1) above (i.e., it joins with at least one atom which is not the image of a partially satisfiable atom that joins with P on z in the same argument positions), then all atoms in the connected component of $grf(\psi)$ that P is in, are partially satisfiable for h' (which means they use extensions of h').*

THEOREM 4. *A set S of atoms is partially satisfiable per Def. 1, iff every atom in S is partially satisfiable per Def. 2.*

PROOF. It is not hard to see that cases 1(a), 1(b) and 2 of Def. 2 correspond to the same cases of Def. 1. However Def. 2 considers "atomic" satisfying homomorphisms over single atoms. We need to make sure that these homomorphisms can be unified to construct one from the entire set of partially satisfiable atoms (i.e., h' of Def. 1). Since every atomic satisfying homomorphism in Def. 2 extends h (that maps the antecedent variables to B,) we really need to examine only existential variables. Notice that all partial satisfiable atoms (per Def. 2) that share existential variables need to fall into the same case of Def. 2. For partially satisfiable atoms in the constraint, that share existential variables and fall in case 2 of Def 2, their satisfying homomorphisms agree and are essentially the same, since the predicates belong in the same connected component of $grf(\psi)$. For partially satisfiable atoms, that share existential variables, and fall in case 1 of Def 2, case 1(c) takes care that their homomorphisms agree on their common values. □

With respect to Def. 2, we define an applicable homomorphism, the frugal chase step and the frugal chase similarly to Def. 1.

To illustrate Def. 2, we run the frugal chase on a view (instead of an instance). For views, distinguished variables are interpreted as constants in our homomorphisms, for both the standard and the frugal chase. Consider the following non-decomposable constraint:

c_5: $P(x, v), R(v, t) \rightarrow P_1(x, y, z), P_2(y, w)$ and view
S_5: $S_5(x_1) \rightarrow P(x_1, v_1), R(v_1, t_1), P_1(x_1, y_1, z_1), P_2(x_1, w_1)$.

The standard chase algorithm run with c_5 on S_5 will produce:

$S_5{}'(x_1) \rightarrow P(x_1, v_1), R(v_1, t_1), P_1(x_1, y_1, z_1), P_2(x_1, w_1),$
$P_1(x_1, y, z), P_2(y, w)$

Nevertheless a shorter, equivalent mapping per our chase is:

$S_5{}''(x_1) \rightarrow P(x_1, v_1), R(v_1, t_1), P_1(x_1, y_1, z_1), P_2(x_1, w_1), P_2(y_1, w)$

When considering whether $P_1(x, y, z)$ in c_5 is partially satisfiable, we notice that y and z can only map to don't care variables in S_5, namely y_1 and z_1 respectively (hence case 1(b) of Def 2 trivially applies). Case 1(c) of Def 2 dictates that we can partially satisfy $P_1(x, y, z)$ as long as y and z are not being mapped to different variables than y_1 and z_1, in some other atom's satisfying homomorphism. This means that $P_2(y, w)$ in c_5 cannot be partially satisfiable when $P_1(x, y, z)$ is, since in order to partially satisfy

$P_2(y, w)$ we would need to map y to a different variable, namely x_1. In fact if we examine $P_2(y, w)$ for partial satisfaction before we examine $P_1(x, y, z)$, we find another reason for which $P_2(y, w)$ cannot be partially satisfiable: variable y can only map to the distinguished variable x_1 in S_5 and, per case 2, we check whether this mapping can be extended to cover the atoms of c_5 joining (directly or indirectly) with the existential variables of $P_2(y, w)$, in effect $P_1(x, y, z)$. Such an extension cannot happen as for P_1 in c_5, y has to map to y_1 in S_5, rather than x_1. Hence, $P_2(y, w)$ in c_5 cannot be partially satisfiable for S_5; only $P_1(x, y, z)$ can and P_2 needs to be added explicitly, when applying the rule.

The frugal chase is applicable to all cases of TGD languages in which the standard chase applies. It yields smaller universal solutions in data exchange, smaller chased mappings in data integration (which lead to smaller rewritings using these mappings), smaller database instances in incomplete databases, and smaller chased queries for query containment (ucq containment under constraints can sometimes be done by chasing one ucq and relying on classic containment [22, 25]). However, our chase does not produce a minimal solution, i.e., the *core* [13]. Using the frugal chase, one could explore all combinations of partially satisfiable atoms and keep the maximum set, but even then, pre-existing redundancies in the instance or the constraints are not accounted for.

5. USING A COMPACT CHASE FOR QUERY REWRITING UNDER CONSTRAINTS

In this section, we present our approach to compute the maximally-contained UCQ rewriting of a target query under target LAV weakly-acyclic dependencies and LAV views (i.e., the *perfect* rewriting [4]), using the frugal chase. First, we show (Th. 5) that our algorithm computes the certain answers (this is the equivalent of Th. 1 [1] for the frugal chase). Second, we describe a graph representation for queries, mappings (views), and now constraints, that we introduced in GQR [24]. This representation allows us to compactly represent common subexpressions in the views and constraints, and efficiently reason with them. Third, we briefly describe the GQR query rewriting approach. Finally, we describe a compact graph-based version of the frugal chase that radically improves the compilation of the constraints into the (graph representation of) the views.

THEOREM 5. *Given a query Q on schema T, a set of LAV schema mappings $\mathcal{V} = \{V_1, ..., V_n\}$, a source instance I under $\mathcal{V}$, and a set of LAV weakly-acyclic TGD constraints Σ, the set of certain answers $certain(Q, I)$ is computed by the UCQ maximally-contained rewriting of Q using $\{V_1{}', ..., V_n{}'\}$, where each $V_i{}'$ is produced by running the frugal chase on V_i using Σ.*

PROOF. Consider the set of views $\mathcal{V}'' = \{V_1'', ..., V_n''\}$, which are taken by running the *standard* chase on the consequent of V_i using Σ. We know by Th. 1 that a maximally-contained rewriting of Q using $\mathcal{V}''$ (denoted $MCR(Q, V'')$) produces the certain answers of the query. The view sets $\mathcal{V}''$ and $\mathcal{V}'$ are equivalent since for every view in one set there is an equivalent one in the other, by Th. 3. Two equivalent sets of views produce equivalent maximally contained rewritings, hence in our case $MCR(Q, V'') \equiv MCR(Q, V')$. This means that for each conjunctive query $r'' \in MCR(Q, V'')$ there is a $r' \in MCR(Q, V')$, s.t. $r'' \subseteq r'$. Let a tuple $t \in certain(Q, I)$ and so $t \in MCR(Q, V'')(I)$ and in particular $t \in r''(I)$. Then $t \in r'(I)$ and hence $t \in MCR(Q, V')$. So every certain answer is computed by $MCR(Q, V')$. Symmetrically, for any tuple t we obtain from $MCR(Q, V')$, with the same reasoning it holds that $t \in MCR(Q, V'')$ and hence is a certain answer. $\square$

5.1 Graph Modeling

We represent queries, mappings, and constraints as graphs (extending [24]). Predicates and their arguments map to graph nodes. Predicate nodes are labeled with the name of the predicate, and are connected through edges to their arguments. Shared variables between atoms result in shared variable nodes, directly connected to predicate nodes. Edges have integer labels that represent the argument positions. We discard variables' names, since to decide on a partial satisfiability and rewriting we only need the type (distinguished or existential) of the variables involved. For example, Figs. 3(a), (b) and (e) show the graph representation of a query q (q(d,c)$\leftarrow D$(d), $DPFC$(d,z,c)), LAV views S_1, S_2 and S_3, and constraint c_1 (cf. Sect. 2), respectively. Distinguished variable nodes are depicted with a circle, and existential ones with the symbol $\otimes$.

Our algorithms consist of mapping subgraphs of the query/constraints to subgraphs of the views. The smallest subgraphs we consider consist of one central predicate node and its (existential or distinguished) variable nodes. We call these primitive graphs *predicate join patterns* (PJs). Fig. 3(d) shows all PJs for sources S_1, S_2 and S_3. Because the join conditions for a particular PJ within each view are different, some "bookkeeping" is needed to capture these joins. For each variable node, we record the joins it participates in across the views in a *"joinbox"*. Figure 3(c) shows the joinbox for the second variable of the TP (TreatsPatient) PJ, which records that this variable joins with the first argument of HCD in S_1 and in S_3.

A critical feature of this representation is that the graph patterns of predicates repeat in multiple views, so we compactly represent all the occurrences of a pattern across different views with one PJ. This has a tremendous advantage; mappings from a query/constraint PJ to a view one are computed just once instead of every time this predicate is met in a view. For example, the 6 predicates in the sources of Fig. 3(b) correspond to only 4 PJs, in Fig. 3(d). PJs can be constructed straightforwardly in polynomial time by scanning each LAV view and its joins, and hashing the different patterns encountered. We also index the patterns to retrieve them efficiently.

5.2 Graph-based Query Rewriting (GQR)

In its original version [24], GQR has a pre-processing/off-line phase, where it extracts the PJs from the views and indexes them.

At query time, GQR (cf. [24] for a detailed description) processes the user query one subgoal at-a-time. It retrieves the view PJs that match each query subgoal and incrementally combines the retrieved view PJs to form larger subgraphs that cover larger portions of the query, building the maximally-contained rewriting incrementally. For example, given the query in Fig. 3(a), and the PJs in Fig. 3(h), Fig. 4(a) shows GQR retrieving the PJs corresponding to query predicates D and DPFC, and combining them into a single graph (Fig. 4(c)). Since the combined graph covers the query, the process terminates and outputs the logical rewritings (Fig. 4(c)).

In this paper, GQR takes as input not the original PJs from the sources, but the PJs chased with the target constraints using the compact frugal chase described next. As discussed, we chase the LAV wa-TGDs into the mappings and reduce the problem to query rewriting (without constraints) using the chased views. Fig. 3(h) shows the view PJs (resulting from the frugal chase) of views S_1', S_2' and S_3' with constraint c_1 (from Section 2). The overall result is an efficient algorithm for query rewriting under constraints.

5.3 Compact frugal chase

This section presents our compact frugal chase implementation for chasing a set of LAV mappings using a set of LAV weakly acyclic dependencies. Instead of considering every view subgoal, our compact frugal chase considers the distinct patterns (PJs) that

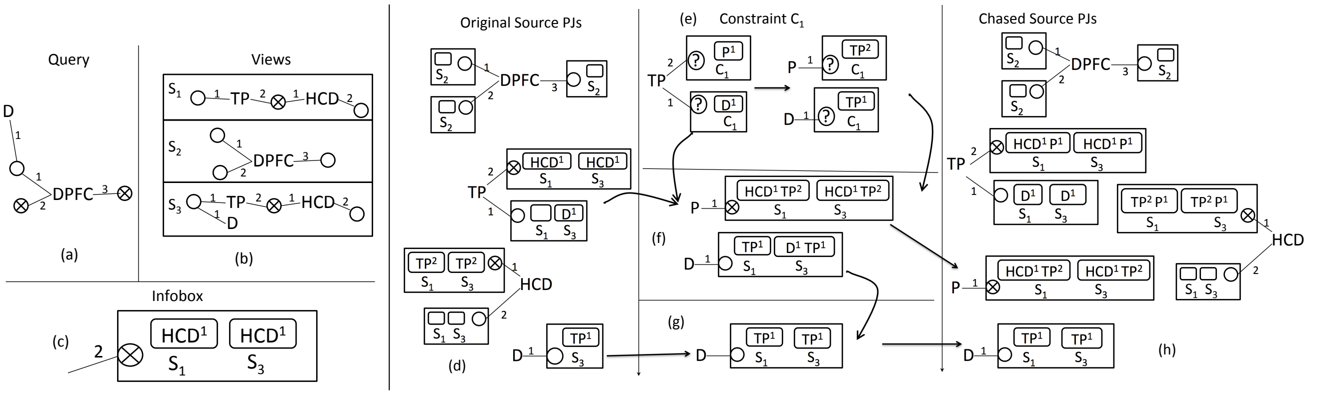


Figure 3: (a) Query q. (b) Sources S_1-S_3 as graphs. (c) Joinbox for a variable node. (d) View PJs for S_1-S_3 with their joinboxes. (e) Constraint c_1 as a graph. (f) PJs resulting from chasing PJ TreatsPatient (TP) with the constraint $c1$. (g) Merging existing and inferred PJs. (h) Chased view PJs.

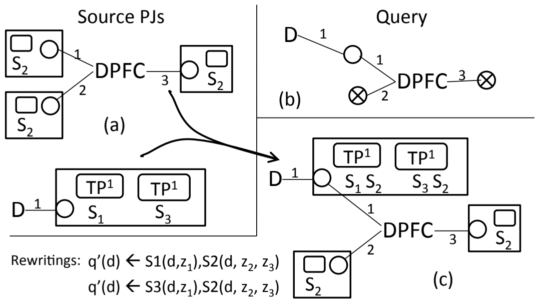

Rewritings: q'(d) ← S1(d,z1),S2(d, z2, z3)
q'(d) ← S3(d,z1),S2(d, z2, z3)

Figure 4: Graph-based Query Rewriting.

all views contain. We start by finding mappings of the antecedent to atomic PJs, that repeat themselves across views and hence compactly represent pieces of multiple views (which imply the same constraint antecedent). Finding a homomorphism to such a pattern means triggering the constraint for all the views represented by this pattern. Then we recursively map all partially satisfiable subgoals of a rule's consequent, to the compact graph representation of these atoms in the mappings. When we apply the chase step we add compact consequent patterns (constraint PJs) that represent the addition of a predicate to multiple views simultaneously. We end up in a compact PJ representation of the chased mappings, shorter than the standard chased ones, which leads to smaller yet equivalent rewritings of the input queries using these mappings. As noted, the output of our compact chase algorithm is using the same PJ representation as its input, and is particularly tailored for GQR [24], improving our solution even more. Our initial implementation does not include constants in the queries, the views or the constraints.

Alg. 1 executes all constraints repeatedly, until it converges and stops adding new predicates. It takes as inputs the view PJs, constructed and indexed, representing our original set of views, and the set of constraints. The output of the algorithm is the set of chased PJs. The algorithm first maps the antecedent predicate to a view PJ in order to trigger the constraint for all views in this view PJ. Then, it examines each consequent's predicate, for all applicable views triggering the constraint; some existing view PJ (that matches a consequent PJ's pattern) might contain some of the relevant views and might prove the corresponding consequent predicate already partially satisfiable. For the applicable views that are left not satisfying a consequent predicate, we will have to add a copy of the consequent PJ as a new view PJs to our index (our set of view PJs will now correspond to the longer, frugal-chased mappings). Using this control flow, Alg. 1 is our high-level algorithm and keeps track of the constraints, and the PJs that satisfy them, for specific views. To check if a specific view PJ is the image of a constraint PJ, for one of the views it contains, it calls upon Alg. 2, which in turn checks the cases of Def. 2, by calling Alg. 4; the latter iterates over connected components of the views or the constraint respectively. Algorithm 4 calls Alg. 3 which, after examining whether a constraint and a view PJ have been processed before, calls again Alg. 2. Hence, algorithms 2, 4, and 3 are mutually recursive.

In line 2, Alg. 1 iterates over all constraints, and in line 3 finds all the different view PJs that match the antecedent (denoted PJ^V_{ant}), for a specific constraint. For each PJ^V_{ant}, line 5 constructs new PJs for all constraint consequents. For each consequent PJ, "initialization" means the following (pseudo-code omitted due to space):
1) Depending on the pattern of PJ^V_{ant}, the variable nodes in a consequent PJ become distinguished or existential (initially constraint node types are undetermined). Fig. 3(f) shows that when we trigger constraint c_1 for the antecedent PJ for TP, the variables of the constraint (originally of unknown (?) type) take a specific type. We change to distinguished all constraint variable nodes that have been "unified" with distinguished view variables; the rest are existential.
2) For all views triggering the constraint ($ViewSet$), we include joinboxes in the consequent PJ. Such joinboxes include all joins to other consequents (which eventually become part of every view), as well as the joins inherited from any variable shared with PJ^V_{ant}.

In effect, when we trigger constraint c_1, in Fig.3(e), for the view PJ TP in Fig.3(d), after line 5 we have constructed the two consequent PJs shown in Fig.3(f), which capture the addition of predicates "P" and "D" in the views they contain. Subsequently we need to examine if for some of these views there are pre-existing PJs satisfying the consequent PJ's predicates. To check this, for every consequent PJ, $PJ^C_{c_i}$, we query our index for view PJs that capture its pattern (it could be a "more" general pattern, i.e., one that has distinguished variables in places that the consequent PJ has existential ones) (lines 6-7). Retrieving such a view PJ, we need to examine it for all views common to the "triggering" views (line 8). The main method to check whether the predicate of $PJ^C_{c_i}$ is partially satisfied by a relevant view s in a view PJ, $PJ^V_{c_i}$, is $isPJSatisfiable$ (line 16), which is a recursive implementation of Def. 2. Due to it's recursive nature, some previous calls of $isPJSatisfiable$ might have traversed the connected component of the constraint and might have actually already memoized the specific combination, of PJs and view currently in question, as being "satisfiable" or "unsatisfiable".

If we find this combination satisfiable, we avoid calling the method again (line 9), and we call $updateJoinboxes$ (line 10) which updates several join pointers on our view and constraint PJs. It first transfers the joinbox information for the specific view s from the

Algorithm 1 Compact Frugal Chase

```
Input: An indexed set of view PJs, a set of LAV-WA constraints $\Sigma$
Output: Chased PJs after the frugal chase converges for $\Sigma$
1: while Index of view PJs changes do
2: for all $\sigma \in \Sigma : P(\vec{x},\vec{y}) \to C_1(\vec{x},\vec{z}),...,C_n(\vec{x},\vec{z})$ do
3:    for all $PJ^V_{ant} \leftarrow$ antecedent PJ in views do
4:       $ViewSet \leftarrow$ views in $PJ^V_{ant}$ not used to trigger $\sigma$
5:       initialize constraint PJs for $ViewSet$
6:       for all constraint PJs $PJ^C_{c_i}$ do
7:          for all $PJ^V_{c_i} \leftarrow$ view PJs capturing pattern $PJ^C_{c_i}$ do
8:             for all view $s$ common to $ViewSet$ and $PJ^V_{c_i}$ do
9:                if $PJ^V_{c_i}$ is marked as satisfiable for $\sigma$,$PJ^C_{c_i}$ and $s$ then
10:                  $updateJoinboxes(PJ^V_{ant},PJ^V_{c_i},PJ^C_{c_i},\sigma,s)$
11:                  if $PJ^C_{c_i}$ "satisfied" for all views in $ViewSet$ then
12:                     continue to the next consequent PJ (line 6)
13:                  continue (line 8)
14:               mark $PJ^V_{c_i}$ as satisfiable for $\sigma$, $PJ^C_{c_i}$ and $s$
15:               State $ST$ ←UNDECIDED
16:               if $isPJSatisfiable(PJ^V_{c_i},PJ^C_{c_i},s,ST)$ then
17:                  $updateJoinboxes(PJ^V_{ant},PJ^V_{c_i},PJ^C_{c_i},\sigma,s)$
18:                  if $PJ^C_{c_i}$ has been "satisfied" for all views in $ViewSet$ then
19:                     continue to the next consequent PJ (line 6)
20:                  continue (line 8)
21:               else
22:                  mark $PJ^V_{c_i}$ as unsatisfiable for $\sigma$, $PJ^C_{c_i}$ and $s$
23:          add $PJ^C_{c_i}$ to Index, update joinboxes of affected PJs
```

consequent to the view PJ. Fig. 3(g) shows that the pre-existing PJ for Doctor in Fig. 3(d) already satisfies the Doctor predicate for S_3, and hence S_3 will be deleted from the consequent PJ of Fig. 3(f) (an additional optimization of our algorithm, shown in the figure, is that if the Doctor PJ of Fig. 3(f) remains unsatisfiable until the end, for some views (e.g., S_1), instead of adding it to our index as an additional PJ we can merge it with an existing one (Fig. 3(g))). Moreover $updateJoinboxes$ updates the joinboxes of all other consequent PJs, s.t. if they are not satisfied and end up as new PJs in our index, their joins already point to $PJ^V_{c_i}$ for this predicate and view.

If after $updateJoinboxes$, which has just deleted the satisfied view from the consequent PJ, the consequent PJ, $PJ^C_{c_i}$, becomes empty, this means that the corresponding consequent predicate is partially satisfiable by existing view PJs for all applicable views. Hence, in line 12, the algorithm jumps to the next consequent PJ. Otherwise, in line 13, the algorithm goes on examining the next applicable view s for $PJ^V_{c_i}$ and $PJ^C_{c_i}$, i.e. it jumps to line 8. If we have no prior information for s, $PJ^C_{c_i}$ and $PJ^V_{c_i}$ we call $isPJSatisfiable$, in line 16. A global variable ST (initially set "undecided" at line 15) maintains which of the two cases of Def. 2 we are currently checking. If we find a partial satisfaction, lines 17-20 repeat the steps described in lines 10-13. Since our algorithms visit joined predicates recursively, we wouldn't like to fall in an infinite cycle and visit the same PJs again for the same view, such as the ones currently considering. In line 14, starting in good faith, we set our pair of PJs and view as already visited and satisfiable, in order to avoid an infinite recursion. If the pair fails to prove partial satisfiability we reverse this decision in line 22. At the end of Alg. 1 (line 23), we will add to our index each consequent PJs with the information for the views left in it (i.e., those views not partially satisfying the consequent predicate). In Fig. 3(f) both sources remain in the consequent PJ for Patient, and this entire PJ is added to our output (Fig. 3(h)). If a new PJ gets added to our index, all view PJs containing the same views (and joining with the new predicate) update their joinboxes accordingly (Fig. 3(h)).

Algorithm 2 isPJSatisfiable($PJ^V_{c_i}$,$PJ^C_{c_i}$,s, ST)

```
Input: A prexisting view PJ $PJ^V_{c_i}$, a constraint PJ $PJ^C_{c_i}$, view $s$, the state $ST$ of this recursion so far
Output: true if the predicate in $PJ^C_{c_i}$ for view $s$ is partially satisfiable in $PJ^V_{c_i}$
1: if $PJ^V_{c_i}$ is marked unsatisfiable for $\sigma$, $PJ^C_{c_i}$ and view $s$ then
2:    return false
3: for all $V_k \leftarrow$ node on edge $k$ of $PJ^V_{c_i}$ do
4:    $joins_{V_k} \leftarrow$ the joinbox joins for $s$ from $V_k$
5:    $C_k \leftarrow$ node on edge $k$ of $PJ^C_{c_i}$
6:    $joins_{C_k} \leftarrow$ the joinbox joins in $C_k$
7:    if $C_k$ is an antecedent variable in the constraint then
8:       if joins with antecedent in $joins_{C_k} \nsubseteq joins_{V_k}$ then
9:          return false
10:   else
11:      if $V_k$ is a distinguished variable then
12:         if ST == 1 then
13:            return false
14:         $ST \leftarrow 2$
15:         if $!checkCase2(joins_{V_k},joins_k,s,ST)$ then
16:            return false
17:      else
18:         switch (ST)
19:            case(2):
20:               if $!checkCase2(joins_{V_k},joins_{C_k},s,ST)$ then
21:                  return false
22:            case(1):
23:               if $!checkCase1(joins_{V_k},joins_{C_k},s,ST)$ then
24:                  return false
25:            case(UNDECIDED):
26:               $ST \leftarrow 2$
27:               if $!checkCase2(joins_{V_k},joins_{C_k},s,ST)$ then
28:                  $ST \leftarrow 1$
29:                  if $!checkCase1(joins_{V_k},joins_{C_k},s,ST)$ then
30:                     return false
31:         if ST == 1 then
32:            for all $PJ^C_{c_j} \leftarrow$ joined predicates in $joins_{C_k}$ do
33:               if $PJ^C_{c_j}$ is marked as satisfiable for $s$, but maps $V_k$ to a different variable than $C_k$ then
34:                  return false
35: return true
```

Checking partial satisfiability. In order to check whether $PJ^V_{c_i}$ can partially satisfy $PJ^C_{c_i}$ for s, Alg. 2 is called. This returns false if at some other point this combination has been marked as unsatisfiable (lines 1-2). Dictated by Def. 2, the partial satisfiability check happens on a variable per variable basis (line 3), for the specific view s, and Alg. 2 returns true if no variable check fails (line 35).

If the constraint variable, C_k, is an antecedent variable, we only need to check that the corresponding view PJ variable, V_k, joins with the antecedent view PJ in the positions that C_k joins with the antecedent constraint PJ (lines 7-9) (i.e., the variables satisfy the constraint antecedent homomorphism). If C_k, is existential, but V_k is distinguished we have to be in case 2 (lines 11-16). We check this by calling, in line 15, $checkCase2$ (shown in Alg. 4) which essentially checks that the joined predicates, to the constraint variable, can be mapped to some of the joined predicates of V_k. This ends up in recursively considering the connected component of the constraint that C_k is in. On the other hand, $checkCase1$, omitted due to space but almost identical to Alg. 4 (just the outer for loops are reversed), considers that all joined predicates of V_k, i.e., the corresponding connected component of the view, are images of constraint predicates joining with C_k.

If V_k is existential then depending on the current state (lines 18-24), we will check to make sure that $PJ^V_{c_i}$ can still partially satisfy

Algorithm 3 joinsSatisfiedRecursively(jd_V, jd_C, s, ST)

```
Input: Join description $jd_V$ of view PJ variable, join description
    $jd_C$ of constraint variable, source $s$, state $ST$ of the recursion
Output: true if the joined predicate described by $jd_V$ is the satis-
    fying homomorphism's image of the joined predicate $jd_C$
 1: if $jd_V == jd_C$ //Joins are to the same predicate name on the
    same position then
 2:     $neighborVPJ \leftarrow$ joined PJ in view described by $jd_V$
 3:     $neighborCPJ \leftarrow$ joined PJ in constraint described in $jd_C$
 4:     if $neighborVPJ$ has been marked as satisfiable for
        $neighborCPJ$ for view $s$ then
 5:         return true
 6:     if $neighborVPJ$ is marked as unsatisfiable for
        $neighborCPJ$ for source $s$ then
 7:         return false
 8:     mark $neighborVPJ$ as satisfiable for $neighbourCPJ$
        for source $s$
 9:     if $isPJSatisfiable(neighborVPJ,neighborCPJ,s,ST)$
        then
10:         return true
11:     mark $neighborVPJ$ as unsatisfiable for $neighbourCPJ$
        for view $s$
12: return false
```

$PJ^C_{c_i}$, in accordance to the corresponding case of the definition. If ST is "undecided" we will check both cases (lines 25-30). After these checks and if we are left in the state of case 1, we test whether case 1(c) of Def. 2 is satisfied, by checking in our memoization structures to see whether the same constraint variable has been mapped somewhere else in the same view and if so we fail (lines 31-34). Algorithms 4 and method $checkCase1$ call Alg. 3 which ends up calling $isPJSatisfiable$ recursively for the joined predicates of our variables, called neighborPJS. Alg. 3 marks the PJs as satisfying (line 8) to avoid an infinite recursion (line 4 guarantees that); this marking is reversed in line 11 if proven wrong.

Algorithm 4 checkCase2($joins_{V_k}$, $joins_{C_k}$, s, ST)

```
Output: true if every joined predicate in $joins_{C_k}$ is partially sat-
    isfiable with a predicate in $joins_{V_k}$
1: for all join descriptions $jd_C$ in $joins_{C_k}$ do
2:     for all join descriptions $jd_V$ in $joins_{V_k}$ do
3:         if $!joinsSatisfiedRecursively(jd_V,jd_C,s,ST)$ then
4:             continue outer for (line 1)
5:     return false //there is one join in constraint not satisfied by
       any join in source
6: return true
```

Alg. 2, implements exactly definition 2 for view s and the predicates in $PJ^V_{c_i}$ and $PJ^C_{c_i}$. In order to prove that our compact chase outputs PJs which capture the frugal chased views, we should point out that the algorithm always terminates since: 1) marking the PJs considered avoids infinite recursion, hence every consequent PJ gets satisfied or added, 2) once finished processing a constraint we do not trigger it again for the same predicate and 3) since our constraints are chase-terminating, and we proved equivalence of the frugal chase to the standard one, we cannot be triggering constraints indefinitely, as at some point we will be finding all consequent predicates partially satisfiable and stop adding new ones. Note that our implementation finds a partially satisfiable set by keeping the first partially satisfiable atom it discovers, together with all those compatible to it. Summarizing, the following holds:

THEOREM 6. *Given a set of view PJs representing our original LAV mappings, and a set of LAV-WA TGD constraints, Alg. 1 always terminates and produces a set of PJs which represent the set of mappings produced after one runs the frugal chase on each one of the view formulas using the constraints.*

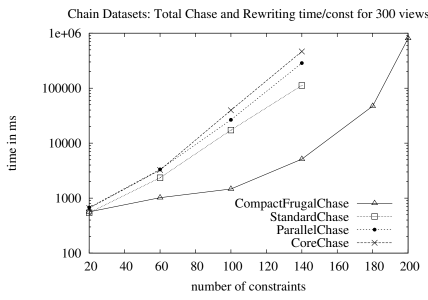
6. EXPERIMENTAL EVALUATION

To evaluate our approach we compare our compact frugal chase algorithm to the standard chase, as well as the parallel and the core chase, in the context of compiling LAV wa-TGD constraints into LAV mappings. First, we run our compact graph chase on a set of conjunctive LAV mappings using a set of LAV weakly acyclic constraints, feeding the resulting PJs to GQR [24] to rewrite a conjunctive query (using the PJs representing the chased mappings). Second, we have implemented and run the standard, parallel and core chase algorithms in order to compile the same sets of constraints into our set of mappings, as in [1]. We compute PJs for the standard, parallel and core chased mappings. In all cases the chased PJs are inputed directly to GQR for query rewriting. Our compact chase outperforms the standard and the core chase by close to 2 and 3 orders of magnitude resp., while our output remains very close to the core. For producing our queries, constraints, and views, we wrote a random query generator, extending the one used in Minicon [31], which is highly customizable as we discuss next. In fact, it can generate queries and views (and constraints) which capture most cases of the mapping scenarios identified in [3] (note that our prototype implementation, however, does not consider constants).

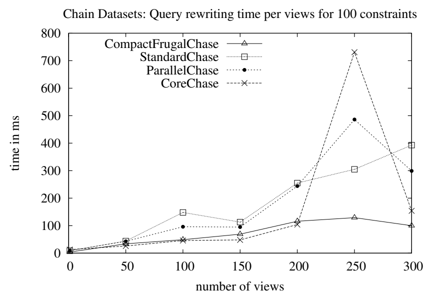
6.1 Chain queries, constraints and views

Initially we generated 20 datasets of chain queries, chain constraints and chain views. We used a space of 300 predicates to generate our LAV weakly-acyclic constraints and views with each constraint/view having 5 predicates joined on a chain. The first atom in our constraints is the antecedent. Each predicate has length 4 and each constraint/view can have up to 3 repeated predicates. Each view has 4 distinguished variables. Additionally in order to get more relevant constraints to views, we generated 10% of our constraints from a smaller subspace of our space of predicates, of size 60. Also, we constructed around 10% of our views by taking one constraint and dropping one of its atoms, causing our constraints to most likely have all atoms except this one partially satisfiable on these views (unless they map their existential variables to distinguished view variables causing case 2 of Def. 2). Lastly, we generated our 20 queries by randomly selecting 3 constraint antecedent atoms and one of these extra non-partially satisfiable atoms that the constraints contain; we do so as to not penalize the standard and parallel chases by querying redundant atoms that they will add (rather, the "extra" atom that the query contains will be probably added by all chase algorithms, including the frugal).

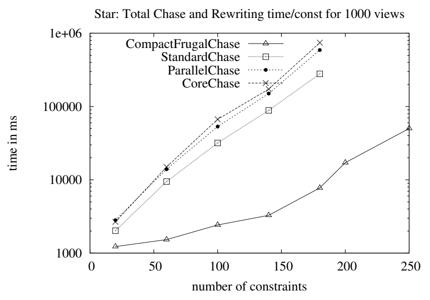
We run our experiments on a cluster of 2GHz processors each with 2Gb of memory. We allocate to each processor 300 views and 300 constraints and one hour of wall time for that job to be finished. Due to the density of our data in this setting, most of the standard/parallel and core chase runs died after reaching 100 constraints (only 7 made it to 140), while for the compact chase all of them reached 140 constraints and 14/20 completed with 180 constraints, which indicates that the compact chase scaled almost twice as much. Figure Fig. 5(a) shows the average total time for running the chases and rewriting the queries (note that the figure averages over the successful queries at each point as discussed above). As seen in the figure, the compact algorithm outperforms the standard and parallel chases by close to two orders of magnitude and the core by close to three. Moreover as Fig. 5(b) shows, the number of PJs that the compact chase produces for 100 constraints, for which all queries succeeded in all frameworks, is the same as the core and is consistently less than the standard/parallel as the number of views increases (for 100 constraints and 300 views the compact chased mappings have 5895 predicates while the standard chased ones have 6232). In fact, the frugal chase computes almost the



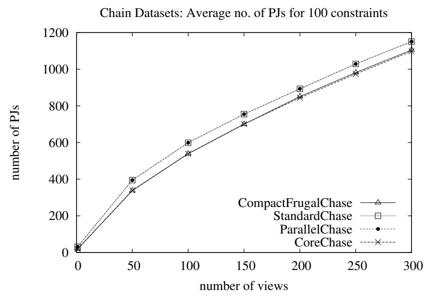
(a)



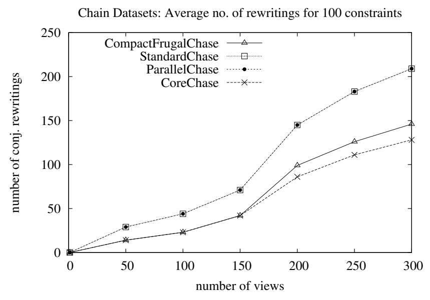
(c)



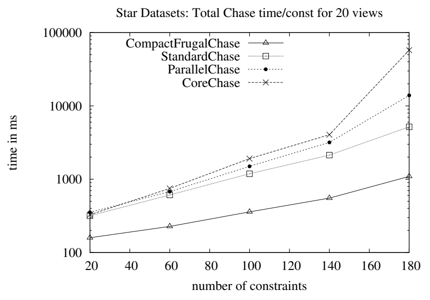
(e)



(b)



(d)



(f)

Figure 5: (a) Total time for 20 chain queries, chasing 300 chain views with up to 200 constraints. (b) Number of PJs produced. (d) Number of rewritings GQR produces, and (c) query rewriting time. (e) Average time of chasing 1000 and (f) 20 star views, with up to 250 star constraints.

same output as the core chase, and this is done very efficiently without minimization. This leads to (equivalent but) fewer conjunctive rewritings for both the frugal and the core chase as output of the query rewriting problem as Fig. 5(d) shows. In particular, for 300 views and 100 constraints our system produced around 30% less query rewritings than the standard/parallel chases, and only 13% more than the core chase. Lastly, the sets of PJs in Figure 5(b) are rather small for GQR, so it reformulates the queries extremely fast (in miliseconds). Fig. 5(c) shows that the reformulation time for the compact chased PJs (and the core chased PJs) is considerably (more than 3x) faster than the standard/parallel chased ones, with the gap between them increasing with the number of views (the core time in Fig. 5(c) for 250 views seems to be an outlier). A side note of our setting is that the parallel chase, which is just a specific ordering on the standard chase execution steps, is slower than the standard chase (see related work for the relevant discussion).

6.2 Star constraints and views

For our second experiment we evaluated our algorithm when it degenerates to the standard chase, and has the same output, i.e. no predicate is partially satisfiable. We did not measure query rewriting; since both compact and standard chase produce the same chased mappings in the form of PJs, GQR takes the same time to rewrite these for any query, yielding identical rewritings.

We created a space with 500 predicate names, out of which we populate our views and constraints. Each atom has 5 variables and each view/constraint can have up to 3 occurrences of the same predicate. In each formula, one predicate is the "star" joining with all other predicates which don't join directly to each other. Using this setting we created 20 datasets of 1000 views and 300 constraints. Each formula has 5 predicates (one of these five predicates in a constraint is the antecedent). Additionally each view has 5 distinguished variables, choosing most of them from the star predicate; this fact essentially introduces distinguished variables in the connected component of the view and obliges our algorithm to fall in case 2 of our definition, i.e., map the constraint consequent in its entirety or not at all (since the consequents are "shorter" than the source descriptions). Since the space of predicates is rather sparse this setting causes the constraints to be unsatisfiable. Moreover it is also difficult for this dataset to produce redundant predicates. This means that the minimization the core chase runs is wasted time.

Each dataset is tested for up to 1000 views and 300 constraints and each processor was allocated a specific number of constraints, scaling from 20 to 300, and all the 1000 views, and we enforced 1 hour wall time. Fig. 5(e) shows the average time for the compact chase versus the standard the parallel and the core chase on all 1000 views. Note that for this experiment the size of the output blows up exponentially as the number of constraints grows: the number of predicates goes from around 5,500 in the original mappings to approximately 180,000 for 250 constraints. This exhausts all algorithms: for the compact chase, all experiments run out of time/memory, for 1000 views, between 250 and 300 constraints, while the other chases crash around 180 constraints. As the figure shows, the compact frugal chase scales to around 30% more constraints while being close to two order of magnitude faster than the standard chase and almost three compared to the core. Additionally note figure Fig. 5(f) which shows the compact chase performance in the same setting but scaling the constraints for only 20 views. Such a small database essentially reduces the advantage of our compact representation: the 20 view formulas generated from our sparse domain, have negligible overlap. Nevertheless our atom oriented implementation of the frugal chase gives a speedup of about one order of magnitude compared to standard/parallel and close to 2 compared to the core. The size of the chased mappings ranges from hundreds for 180 constraints to thousands for 250 constraints.

7. RELATED WORK

Early approaches dealing with relational query rewriting, with no constraints, involve algorithms such as the *bucket* [20], and the *inverse rules* [12]. Modifications of the inverse rules algorithm for full dependencies or functional dependencies where presented in [12]. Koch [23] presented another variation of inverse rules for answering queries using views under GLAV dependencies (for dependencies common to both source and global schemas). A more efficient approach to view-based query answering was proposed in 2001, by the MiniCon [31] algorithm. MCDSAT [5] exhibited a better performance than MiniCon, and is essentially the MiniCon algorithm cast as a satisfiability problem. In this paper we employ GQR [24] which significantly outperformed MCDSAT.

There are approaches to data integration and query answering under constraints for which the chase does not terminate, but rather one has to do query *expansion*, using the constraints. The family of DL-Lite languages [4] is a famous such first-order rewritable class of constraints, which underlies the semantic web language OWL2-QL. Extensions of DL-Lite include flavors of the Datalog+/- family of languages [8]. Query *expansion* rewrites the query using the constraints into a UCQ, before even considering the views or data. Although the LAV wa-TGDs used in this work are both FO-rewritable and chase-terminating, we focus on running the chase on the views without needing to take the query into account, in order to speed up the system's online performance. There are flavors of Datalog+/-, such as the guarded fragment [7], for which the chase does not terminate, but there is a finite part of the infinite chase sufficient for query answering, but again, once a specific query is given. Our frugal chase could substitute the chase algorithms in these fragments.

Chase&Backchase (C&B) [11] is a technique for query minimization and for finding equivalent rewritings, by first chasing all constraints and mappings and then minimizing the chase. Afrati et. al [1] presented an optimized version for finding equivalent rewritings, as well as maximally-contained, as discussed. In this paper we outperformed the latter approach (running the standard chase and then doing query rewriting). Additionally, the frugal chase can replace the chase in the (C&B) algorithm, for finding equivalent rewritings and be even more advantageous (being shorter than the standard chase) for query minimization. Our frugal chase can be used for query containment; in [25] we used the standard chase and a compact graph-based algorithm for UCQ containment.

The chase algorithm has been studied in multiple variations in the recent literature (for overviews see [30, 18]), which guarantee chase termination for one or all instances, under different classes of constraints, producing universal solutions. These include the naive oblivious chase [7], the skolem oblivious chase [28] and the parallel and core chase [10]. Whenever the oblivious and naive chases terminate: 1) the standard chase also terminates and 2) they produce longer solutions than the standard chase [30, 18]. This justifies our choice of optimizing and implementing the standard chase since we focus on producing shorter chased mappings. The parallel chase is just a specific ordering of standard chase step executions; in particular, each parallel chase step decides all applicable constraints on the instance as it has been formed so far and, only after finishing considering all constraints, adds all consequents before going to the next step. Notice that this might penalize the parallel chase as evident by our experiments; the "incremental" addition of consequents that the standard chase performs might prove some "parallel" rules satisfiable, and hence avoid applying them. Each core chase step is a parallel chase step followed by minimization of the produced instance. As its name suggests the core chase leads to a minimal universal solution, i.e., the *core*, as opposed to the frugal chase which leads to smaller but not necessarily minimal solutions. The prize paid however by the core chase is extra minimization (whose equivalent decision problem is NP-hard), as indicated by our experiments. Nevertheless the core chase is more "general" than the standard and the frugal chase since it is a complete algorithm for producing universal solutions under more general classes of constraints [10]. Notice that the "building block" of the core and parallel chases is the standard chase which we could replace with the frugal chase in order to implement these algorithms. This can be very useful as the frugal chase step is not only faster but produces smaller intermediate instances; the minimization step of the core chase would also run faster on such instances. Other data exchange literature that computes the core universal solution, includes [13, 16, 17]. The space of universal solutions (such as our frugal chase) which lie in between the core and the standard chase has been discussed by Henrich et al. [21]. Approaches in [32, 29] construct mappings and constraints that use negation to avoid adding redundant predicates to a solution. We plan to evaluate our approach in these settings and extend our algorithms to produce the core by leveraging our compact graph-based format.

8. DISCUSSION

We have presented several contributions. First, we introduced the *frugal* chase, which produces smaller universal solutions than the standard chase. Second, we developed an efficient compact version of the frugal chase. Third, we used this frugal chase to scale up query rewriting under constraints, for the case of LAV mappings and LAV weakly acyclic constraints. As our experiments show, we gain the additional expressivity of using dependencies very cheap: in essence, our algorithm only pays the cost of relational query answering using our preprocessed views, which due to our optimized, shorter chase, our compact format, and our indexing, becomes very efficient. In future work, we plan to explore extensions of our chase that would compute the core solution, as well as evaluate our system both in data exchange scenarios (i.e., chase instances rather than mappings) and with other chase-terminating cases of constraints, including limited interactions of TGDs and EGDs.

9. ACKNOWLEDGMENTS

This work was supported in part by the NIH through the NIGMS grant: Biomedical Informatics Research Network (U24-GM104203-05), and in part through the NIMH grants: Collaborative Center for Genetic Studies of Mental Disorders (U24-MH068457-12), and SchizConnect:Large-Scale Schizophrenia Neuroimaging Data Mediation & Federation (U01-MH097435-01). We thank the anonymous reviewers for their useful feedback and suggestion to compare against the parallel and core chases.

10. REFERENCES

[1] F. N. Afrati and N. Kiourtis. Computing certain answers in the presence of dependencies. *Inf. Sys.*, 35(2):149-169, 2010.

[2] F. N. Afrati and P. G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *ICDT*, 2009.

[3] B. Alexe, W.-C. Tan, and Y. Velegrakis. STBenchmark: towards a benchmark for mapping systems. *PVLDB*, 1(1):230–244, 2008.

[4] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The dl-lite family and relations. *J. of Artificial Intelligence Research*, 36:1–69, 2009.

[5] Y. Arvelo, B. Bonet, and M. E. Vidal. Compilation of query-rewriting problems into tractable fragments of propositional logic. In *AAAI'06*, pgs 225–230, 2006.

[6] C. Beeri & M. Vardi. The implication problem for data dependencies. *Automata, Languages & Programming*, Springer, pages 73-85, 1981.

[7] A. Cali, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *JAIR*, 48:115–174, 2013.

[8] A. Cali, G. Gottlob, T. Lukasiewicz, and A. Pieris. Datalog+/-: A family of languages for ontology querying. In *Datalog Reloaded*, Springer, pg 351–368, 2011.

[9] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In STOC, 1997.

[10] A. Deutsch, A. Nash, and J. Remmel. The chase revisited. In *PODS*, pages 149–158, 2008.

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

[12] O. M. Duschka, M. R. Genesereth, and A. Y. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1):49–73, 2000.

[13] R. Fagin, P. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM TODS*, 30(1):174–210, 2005.

[14] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89 – 124, 2005.

[15] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational plans for data integration. In *AAAI*, pages 67–73, 1999.

[16] G. Gottlob. Computing cores for data exchange: new algorithms and practical solutions. In *PODS*, 2005.

[17] G. Gottlob and A. Nash. Efficient core computation in data exchange. *Journal of the ACM (JACM)*, 55(2):9, 2008.

[18] S. Greco, C. Molinaro, and F. Spezzano. Incomplete data and data dependencies in relational databases. *Synthesis Lectures on Data Management*, 4(5):1–123, 2012.

[19] J. Gryz. Query rewriting using views in the presence of functional and inclusion dependencies. *Information Systems*, 24(7):597–612, 1999.

[20] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[21] A. Hernich, L. Libkin, and N. Schweikardt. Closed world data exchange. *ACM Transactions on Database Systems (TODS)*, 36(2):14, 2011.

[22] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *J.Comp.&Sys. Sciences*, 28(1):167–189, 1984.

[23] C. Koch. Query rewriting with symmetric constraints. *AI Communications*, 17(2):41–55, 2004.

[24] G. Konstantinidis and J. L. Ambite. Scalable query rewriting: A graph-based approach. In *ACM SIGMOD*, 2011.

[25] G. Konstantinidis and J. L. Ambite. Scalable containment for unions of conjunctive queries under constraints. In *Semantic Web Information Management*, New York, New York, 2013.

[26] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246. ACM, 2002.

[27] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.

[28] B. Marnette. Generalized schema-mappings: from termination to tractability. In *PODS*, pages 13–22, 2009.

[29] G. Mecca, P. Papotti, and S. Raunich. Core schema mappings. In *Proc ACM SIGMOD*, pg 655–668, 2009.

[30] A. C. Onet. *The chase procedure and its applications*. PhD thesis, Concordia University, 2012.

[31] R. Pottinger and A. Halevy. MiniCon:a scalable algorithm for answering queries using views. *The VLDB Journal*, 2001.

[32] B. Ten Cate, L. Chiticariu, P. Kolaitis, and W.-C. Tan. Laconic schema mappings: Computing the core with sql queries. *PVLDB*, 2(1):1006–1017, 2009.

APPENDIX

PROOF OF THEOREM 2. We prove the theorem for Def. 2 since it is equivalent to Def. 1 by Th. 4. The frugal chase step is applicable iff the standard chase step is, by Lemma 1. We first show that there exist homomorphisms that map B' to B'' and vice versa, assuming σ is applicable; which means there is a homomorphism h from the antecedent of σ, say $\phi(\vec{x},\vec{z})$ to B s.t. it cannot be extended over the consequent, $\psi(\vec{x},\vec{y})$. We know by Lemma 1 that there is a conjunct with at least one atom in σ, $\psi_2(\vec{x},\vec{y})$, that is not partially satisfiable for h. If ψ_2 is the entire consequent of the constraint, then the applicable homomorphism by definition will be the same as the standard chase application homomorphism (modulo names of fresh variables). This means that B' and B'' are homomorphically equivalent (in fact, they are isomorphic since there is no essential optimization to the result of the frugal chase step).

We now examine the case where a part of the constraint, say ψ_1 is partially satisfiable, while another part ψ_2 is not. Conjunct ψ_1 can have three categories of variables/constants: (1) Those that are antecedent variables or constants in the constraint, i.e., belong in $\vec{x}$ (we abuse notation and regard that constants are in $\vec{x}$). (2) Those existential variables, $e_{\vec{\psi_2}}$, that (a) fall under case 2 of our definition, i.e., they map to constants/distinguished variables or existential variables that join with a part of the database which is not a partially satisfiable image, together with variables that (b) belong in the same gaifman connected component with the variables in (a). We denote by $e_{\vec{\psi_2}B}$ the images of $e_{\vec{\psi_2}}$ in the database. All atoms containing $e_{\vec{\psi_2}}$ in the constraint should be partially satisfiable, hence the variables of $e_{\vec{\psi_2}}$ exist only in the conjunction ψ_1 in the constraint. (3) Those existential variables that map to variables that only exist in images of partially satisfiable atoms and are not in $e_{\vec{\psi_2}}$, i.e, they are not contained in any "case 2 predicate" above; we denote these variables in the constraint as $e_{\vec{\psi_1}}$. We denote by $e_{\vec{\psi_1}B}$ the images of $e_{\vec{\psi_1}}$ in the database. All atoms containing $e_{\vec{\psi_1}B}$ are images of partially satisfiable atoms, hence $e_{\vec{\psi_1}B}$ exists only in the image of ψ_1 in the database.

Hence, our constraint has the form: σ: $\phi(\vec{x},\vec{y}) \rightarrow \psi_1(\vec{x}, e_{\vec{\psi_1}}, e_{\vec{\psi_2}})$, $\psi_2(\vec{x}, e_{\vec{\psi_1}}, \vec{r_c})$ with $\vec{r_c}$ the rest of the variables belonging solely in ψ_2. In general, our database instance has the form (reusing ϕ, and ψ_1 as the images of the antecedent and the partially satisfiable conjunction, respectively): $\{\phi(\vec{x_B},\vec{y_B}),\ \psi_1(\vec{x_B},\ e_{\vec{\psi_1}B}, e_{\vec{\psi_2}B}),\ \psi_3(\vec{x_B}, e_{\vec{\psi_2}B},\ \vec{r_B},\ \vec{y_B}))\}$ with ψ_3 denoting the rest of the atoms in the database (if any). Terms in $\vec{x_B}$ and $\vec{y_B}$ are the images of $\vec{x}$ and $\vec{y}$ respectively, and $\vec{r_B}$ contains the rest of the terms in the database. Moreover, by definition there is no constraint predicate that contains variables from both $e_{\vec{\psi_1}}$ and $e_{\vec{\psi_2}}$. Hence, we can "break" ψ_1 in the constraint into two parts, one containing $e_{\vec{\psi_1}}$ and one containing $e_{\vec{\psi_2}}$ (exactly one of the two parts might be empty). Our constraint now can be written as σ: $\phi(\vec{x},\vec{y}) \rightarrow \psi_{11}(\vec{x}, e_{\vec{\psi_1}})$, $\psi_{12}(\vec{x}, e_{\vec{\psi_2}})$, $\psi_2(\vec{x}, e_{\vec{\psi_1}}, \vec{r_c})$. Similarly our instance becomes: $\{\phi(\vec{x_B},\vec{y_B})$, $\psi_{11}(\vec{x_B},\ e_{\vec{\psi_1}B}),\ \psi_{12}(\vec{x_B},\ e_{\vec{\psi_2}B}),\ \psi_3(\vec{x_B}, e_{\vec{\psi_2}B}, \vec{r_B}, \vec{y_B})\}$.
The standard chase run on this database with σ will produce the instance: $\{\phi(\vec{x_B},\vec{y_B}),\ \psi_{11}(\vec{x_B},\ e_{\vec{\psi_1}B}),\ \psi_{12}(\vec{x_B},\ e_{\vec{\psi_2}B}),\ \psi_3(\vec{x_B},\ e_{\vec{\psi_2}B}, \vec{r_B}, \vec{y_B}),\ \psi_{11}(\vec{x_B},\ e_{\vec{\psi_1}}),\ \psi_{12}(\vec{x_B},\ e_{\vec{\psi_2}}),\ \psi_2(\vec{x_B},\ e_{\vec{\psi_1}},\ \vec{r_c})\}$.
The frugal chase will produce $\{\phi(\vec{x_B},\vec{y_B}),\ \psi_{11}(\vec{x_B}, e_{\vec{\psi_1}B}),\ \psi_{12}(\vec{x_B},\ e_{\vec{\psi_2}B}),\ \psi_3(\vec{x_B},\ e_{\vec{\psi_2}B}, \vec{r_B}, \vec{y_B}),\ \psi_2(\vec{x_B},\ e_{\vec{\psi_1}B},\ \vec{r_c})\}$. To verify that the two instances are equivalent, the only non-obvious fact is that $e_{\vec{\psi_1}B}$ can map to $e_{\vec{\psi_1}}$. Since case 1 of Def. 2, states that in the same positions that two atoms in the database contain $h'(z)$, the corresponding atoms in the constraint contain z, it means that the restriction of $h'(z)$ on these variables is one-to-one, and the inverse can map $h'(z)$ to z. Hence, the two instances are homomorphically equivalent. Moreover, if the databases are views, the part containing distinguished variables (ψ_{12}) maps to itself and the homomorphisms that prove equivalence are containment mappings. Lastly, the fact that B'' satisfies the constraint, is directly proven by using the applicable homomorphism that we used to construct B''.