Benchmarking the Chase

Michael Benedikt [1] George Konstantinidis [1] Giansalvatore Mecca [2]
Boris Motik [1] Paolo Papotti [3] Donatello Santoro [2]
Efthymia Tsamoura [1]

[1] University of Oxford [2] University of Basilicata [3] Arizona State University

ABSTRACT

The chase is a family of algorithms used in a number of data management tasks, such as data exchange, answering queries under dependencies, query reformulation with constraints, and data cleaning. It is well established as a theoretical tool for understanding these tasks, and in addition a number of prototype systems have been developed. While individual chase-based systems and particular optimizations of the chase have been experimentally evaluated in the past, we provide the first comprehensive and publicly available benchmark—test infrastructure and a set of test scenarios—for evaluating chase implementations across a wide range of assumptions about the dependencies and the data. We used our benchmark to compare chase-based systems on data exchange and query answering tasks with one another, as well as with systems that can solve similar tasks developed in closely related communities. Our evaluation provided us with a number of new insights concerning the factors that impact the performance of chase implementations.

1. INTRODUCTION

The chase [25] is a long-standing technique developed by the database community for reasoning with constraints expressed as a certain kind of logical formulas. When applied to a set of constraints given as *dependencies* (i.e., universal implications possibly containing existential quantification in the conclusion) and a set of facts, the chase extends the facts in a forward-chaining manner to satisfy the dependencies.

The chase has been intensively studied as a theoretical tool; however, over the last decade practical aspects such as developing optimizations of the chase algorithms and building systems based on the chase for various tasks have also been considered (see Section 9 for a list of references). These studies have reported mixed results. On the one side, highly scalable algorithms to chase source-to-target dependencies have been proposed. However, as soon as target dependencies are thrown into the mix, scalability has been achieved only in quite restricted cases. As a consequence, the real performance of chase systems on large sets of complex dependencies and large datasets remains unknown.

This suggest that it is time to evaluate the extent to which computing the chase is practically feasible.

The chase is closely related to and can be seen as a special case of theorem proving calculi such as tableaux and resolution, and it can also be seen as a generalization of standard query evaluation in databases. But while the theorem proving and the database communities have a long history of creating benchmarks and detailed evaluation methodologies (e.g., SMTLib [37] and TPTP [40] in the former, the TPC family [39] in the latter), there is little corresponding infrastructure to support experimental validation of systems such as the chase that combine reasoning and data management.

This paper aims to take a major step in changing this situation. We present a new benchmark for chase systems covering a wide range of scenarios. Since the systems in the literature support different kinds of dependencies, we have created subsets of the benchmark with distinct characteristics in terms of dependency structure and data size.

We use these dependencies to define example tasks for two main applications of the chase: (i) data exchange, which involves materializing an instance of a target schema satisfying a given set of dependencies with respect to an instance of a source schema; and (ii) computing certain answers to conjunctive queries in databases with dependencies.

We then analyze a variety of publicly available systems on our benchmark in order to answer the following questions:

- How do existing chase-related systems fare in absolute terms on these tasks? That is, to what extent can they be considered as proof that the chase-based approaches to solving these tasks are practically feasible?
- Which algorithmic and architectural choices are most critical for the performance of chase-related systems?
- Are there other approaches or other kinds of systems that can perform these same tasks and, if so, how do they compare to tools that use the chase?

Towards answering these questions, we consider a number of systems that implement the chase as a component, including systems motivated from data exchange, data cleaning, query reformulation, and query answering.

We mentioned above that many communities have looked at techniques similar to the chase, and at problems similar to data exchange and query answering. To better understand the connection with the related communities, we also applied our benchmark to systems that are not specifically "branded" as chase systems, but that can nonetheless perform some of the tasks that the chase addresses. In particular, we looked at Datalog engines that support function symbols as they can solve both data exchange and query answering problems, as well as a leading theorem prover that can solve various query answering problems.

Organization. In the rest of this paper, we first present some background about the chase (Sections 2 and 3). Next, we describe our test systems (Section 4), and discuss our testing infrastructure and test scenarios (Section 5). Then,

we present the system comparison results (Section 6), followed by a discussion of the insights gained (Section 7) and the future challenges that emerged from our study (Section 8). Finally, we close with a discussion of the related work and conclusions (Sections 9 and 10). We emphasize that *full details regarding the systems under test, our test infrastructure, and our scenarios are available on the benchmark Web page* (`http://dbunibas.github.io/chasebench`).

2. BACKGROUND

Database basics. Let Const, Nulls, and Vars be mutually disjoint, infinite sets of *constant values*, *labeled nulls*, and *variables*, respectively. Intuitively, constant values are unique; labeled nulls represent unknown values; and variables are used in dependencies and queries. A *value* is a constant value or a labeled null, and a *term* is a value or a variable. We often abbreviate an n-tuple of terms $t_1,\ldots,t_n$ as $\vec{t}$, and we often treat the latter as a set and write $t_i \in \vec{t}$.

A *schema* is a set of *relation names* (or just *relations*), each associated with a nonnegative integer called *arity*. An *instance* I of a schema assigns to each n-ary relation R in the schema a (possibly infinite) set $I(R)$ of n-tuples of values. The *active domain* of I is the set of values occurring in some tuple of some $I(R)$. A *relational atom* has the form $R(\vec{t})$ where $\vec{t}$ is an n-tuple of terms. An *equality atom* has the form $t_1 = t_2$ where t_1 and t_2 are terms. A *fact* is an atom that does not contain variables. An instance can equivalently be seen as a set of relational facts, so we use notation $R(\vec{t}) \in I$ and $\vec{t} \in I(R)$ interchangeably. An atom (resp. an instance) is *null-free* if it does not contain null values.

Term mappings. A *term mapping* σ is a partial mapping of terms to terms; we write $\sigma = \{t_1 \mapsto s_1,\ldots,t_n \mapsto s_n\}$ to denote that $\sigma(t_i) = s_i$ for $1 \le i \le n$. For α a term, an atom, a conjunction of atoms, or a set of atoms, $\sigma(\alpha)$ is obtained by replacing each occurrence of a term t in α that also occurs in the domain of σ with $\sigma(t)$. The *composition* of σ with a term mapping μ is the term mapping $\sigma \circ \mu$ on the union of the domains of σ and μ where $(\sigma \circ \mu)(t) = \sigma(\mu(t))$.

A *substitution* σ is a term mapping whose domain contains only variables and whose range contains only values; moreover, σ is *null-free* if its range contains only constants; finally, σ is a *homomorphism* of a conjunction of atoms $\rho = \bigwedge_i A_i$ into an instance I if the domain of σ is the set of all variables occurring in ρ and $\sigma(\rho) \subseteq I$.

Dependencies and solutions. Semantic relationships between relations can be described using *dependencies*, which come in two forms. A *Tuple Generating Dependency* (TGD) is a logical sentence of the form (1), where $\lambda(\vec{x})$ is a conjunction of relational, null-free atoms whose free variables are contained in $\vec{x}$, and $\rho(\vec{x},\vec{y})$ is a conjunction of relational, null-free atoms whose free variables are contained in $\vec{x} \cup \vec{y}$.

$$\forall \vec{x}\ \lambda(\vec{x}) \rightarrow \exists \vec{y}\ \rho(\vec{x},\vec{y}) \qquad (1)$$

An *Equality Generating Dependency* (EGD) is a logical sentence of the form (2), where $\lambda(\vec{x})$ is a conjunction of relational, null-free atoms over variables $\vec{x}$, and $\{x_i, x_j\} \subseteq \vec{x}$

$$\forall \vec{x}\ \lambda(\vec{x}) \rightarrow x_i = x_j \qquad (2)$$

The left-hand side of a TGD or an EGD (i.e., the conjunction $\lambda(\vec{x})$) is the *body* of the dependency, and the right-hand side is the *head*. A dependency is *linear* if it has exactly one atom in the body. By a slight abuse of notation, we often treat heads and bodies as sets of atoms. For brevity, we commonly omit the leading universal quantifiers in dependencies.

Let I be an instance and let τ be a TGD of the form (1) or an EGD of the form (2). The notion of τ *holding* in I (or I *satisfying* τ, written $I \models \tau$) is given by first-order logic, and it can be restated using homomorphisms as follows. A homomorphism h of $\lambda(\vec{x})$ into I is a *trigger* for τ in I. If τ is a TGD, such h is *active* if no extension of h to a homomorphism of $\rho(\vec{x},\vec{y})$ into I exists; and if τ is an EGD, such h is *active* if $h(x_i) \neq h(x_j)$. Dependency τ is then satisfied in I if no active trigger for τ in I exists.

Let Σ be a set of dependencies and let I be a finite, null-free instance. Instance J is a *solution* for Σ and I if $I \subseteq J$ and $J \models \tau$ for each $\tau \in \Sigma$. Moreover, a solution J for Σ and I is *universal* if, for each solution J' for Σ and I, a term mapping μ from the active domain of J to the active domain of J' exists such that (i) $\mu(c) = c$ for each constant value $c \in$ Const, and (ii) $\mu(J) \subseteq J'$. Solutions for Σ and τ are not unique, but universal solutions are unique up to homomorphism.

Queries. A *conjunctive query* (CQ) is a formula of the form $\exists \vec{y} \bigwedge_i A_i$, where A_i are relational, null-free atoms. A substitution σ is an *answer* to Q on instance I if the domain of σ is precisely the free variables of Q, and if σ can be extended to a homomorphism of $\bigwedge_i A_i$ in I. By choosing a canonical ordering for the free variables $\vec{x}$ of Q, we often identify σ with an n-tuple $\sigma(x_1),\ldots,\sigma(x_n)$. The *output* of Q on I is the set $Q(I)$ of all answers to Q on I.

Answering queries under dependencies. Let Σ be a set of dependencies, let I be a finite, null-free instance, and let Q be a CQ. A substitution σ is a *certain answer* to Q on Σ and I if σ is an answer to Q on each solution J for Σ and I. The task of finding all certain answers to Q on Σ and I is called *query answering under dependencies*, and we often abbreviate it to just *query answering*. The following fundamental result connects universal solutions and query answering: *for each substitution σ and each universal solution J for Σ and I, substitution σ is a certain answer to Q on Σ and I if and only if σ is a null-free answer to Q on J* [13].

The chase. The chase modifies an instance by a sequence of *chase steps* until all dependencies are satisfied. Let I be an instance. A chase step for a TGD τ of the form (1) and a trigger h extends I with facts $h'(A_i)$, where h' is a substitution such that $h'(x_i) = h(x_i)$ for each variable $x_i \in \vec{x}$, and $h'(y_j)$, for each $y_j \in \vec{y}$, is a fresh labeled null that does not occur in I. Moreover, a chase step for an EGD τ of the form (2) and a trigger h *fails* if $\{h(x_i) \neq h(x_j)\} \subseteq$ Const, and otherwise it computes $\mu(I)$ where $\mu = \{h(x_j) \mapsto h(x_i)\}$ if $h(x_i) \in$ Const, and $\mu = \{h(x_i) \mapsto h(x_j)\}$ if $h(x_i) \notin$ Const.

For Σ a set of TGDs and EGDs and I a finite, null-free instance, a *chase sequence* for Σ and I is a (possibly infinite) sequence $I_0, I_1, \ldots$ such that $I = I_0$ and, for each $i > 0$, instance I_i (if it exists) is obtained from I_{i-1} by applying a successful chase step to a dependency $\tau \in \Sigma$, and an active trigger h for τ in I_{i-1}. The sequence must be *fair*: for each $\tau \in \Sigma$, each $i \ge 0$, and each active trigger h for τ in I_i, some $j > i$ must exist such that h is not an active trigger for τ in I_j (i.e., no chase step should be postponed indefinitely). The *result* of a chase sequence is the (possibly infinite) instance

$I_\infty = \bigcup_{i \geq 0} \bigcap_{j \geq i} I_j$. Since EGD chase steps can fail, a chase sequence for a given Σ and I may not exist. Moreover, chase steps can be applied in an arbitrary order so a chase sequence for Σ and I is not unique. Finally, EGD steps are not monotonic (i.e., $I_{i-1} \not\subseteq I_i$ holds when I_i is obtained by applying an EGD step to I_{i-1}), and so I_∞ is not uniquely determined by Σ and I. Nevertheless, each result I_∞ of a chase sequence for Σ and I is a universal solution for Σ and I [13].

A finite chase sequence is said to be *terminating*; and a set of dependencies Σ *has terminating chase* if, for each finite, null-free instance I, each chase sequence for Σ and I is terminating. For such Σ, the chase provides us with an effective approach to computing certain answers to a CQ Q on Σ and I: we compute (any) chase I_∞ of Σ and I, we compute the output $Q(I_\infty)$, and finally we remove all substitutions that are not null-free. Checking whether a set of dependencies Σ has terminating chase is undecidable [11]. *Weak acyclicity* [13] was one of the first sufficient polynomial-time conditions for checking whether Σ has terminating chase. Stronger sufficient (not necessarily polynomial-time) conditions have been proposed subsequently [18, 32].

Data exchange. In relational-to-relational *data exchange* [13], a transformation of an arbitrary instance of a *source schema* into an instance of a *target schema* is described using

- a set Σ_{st} of *s-t* (source-to-target) TGDs where all body atoms use relations of the source schema and all head atoms use relations of the target schema, and
- a set Σ_t of *target dependencies* (i.e., TGDs or EGDs) whose atoms use relations of the target schema.

Given a finite, null-free instance I of the source schema, the objective of data exchange is to compute a universal solution to the set of dependencies $\Sigma = \Sigma_{st} \cup \Sigma_t$ and I. If Σ has terminating chase, then a universal solution can be computed using the chase, and it can be used to answer queries [13].

3. IMPLEMENTING THE CHASE

Algorithm 1, when it terminates, computes the fixpoint of a chase sequence: in each iteration (lines 2–16), for each dependency τ (line 4) and trigger h (line 5), it checks whether h is active (line 6), and, if so, it applies the TGD (lines 8–9) or the EGD (lines 11–13) chase step. Although chase systems may and do depart from these specifics, Algorithm 1 captures essential aspects of all implementations. In the rest of this section we first highlight certain basic issues and then discuss several chase variants. As our experiments show (cf. Section 6), the choice of the chase variant has a major impact on the performance of practical systems, which will allow us to draw in Section 7 some very general conclusions about different implementation strategies.

Substitution h is a trigger for a dependency τ of the form (1) or (2) in an instance I if and only if h is an answer to $\lambda(\vec{x})$ on I; thus, the triggers for τ are determined in line 5 by evaluating $\lambda(\vec{x})$ as a CQ over I. Moreover, if τ is a TGD, h is active if and only if $\gamma(I) = \emptyset$ where $\gamma = \exists \vec{y}\, \rho(h(\vec{x}), \vec{y})$ is a Boolean CQ. Consequently, all chase systems require efficient evaluation of CQs in an instance, which can be realizes in different ways: some systems are built on top of RDBMSs, and some use RAM-based storage.

Evaluating $\lambda(\vec{x})$ in each iteration "from scratch" is very inefficient since it repeatedly considers triggers from all pre-

Algorithm 1 RESTRICTED-CHASE(Σ, I)

```
 1: ΔI := I
 2: while ΔI ≠ ∅ do
 3:   N := ∅,     μ := ∅
 4:   for each τ ∈ Σ with body λ(x⃗) do
 5:     for each trigger h ∈ λ(I) with h(λ(x⃗)) ∩ ΔI ≠ ∅ do
 6:       if trigger h is active for τ and μ(I ∪ N) then
 7:         if τ = ∀x⃗ λ(x⃗) → ∃y⃗ ρ(x⃗, y⃗) is a TGD then
 8:           h' := h ∪ {y⃗ ↦ v⃗} where v⃗ ⊆ Nulls is fresh
 9:           N := N ∪ h'(ρ(x⃗, y⃗))
10:         else if τ = ∀x⃗ λ(x⃗) → x_i = x_j is an EGD then
11:           if {h(x_i) ≠ h(x_j)} ⊆ Const then fail
12:           ω := {min(h(x_i), h(x_j)) ↦ max(h(x_i), h(x_j))}
13:           μ := μ ∘ (ω ∘ μ)
14:   N := μ(N) ∪ (μ(I) \ I)
15:   ΔI := N \ I
16:   I := μ(I) ∪ ΔI
```

vious iterations. Thus, any reasonable chase system should use the *seminaïve* evaluation [1], cf. Algorithm 1. In each iteration, the consequences of chase steps are accumulated in an auxiliary set N and term mapping μ, and line 15 ensures that ΔI contains the newly derived facts from the last iteration. Moreover, at least one atom of $h(\lambda(\vec{x}))$ must be in ΔI (cf. line 5), so each iteration considers only triggers obtained using at least one fact from the last derivation.

The consequences of TGDs are accumulated by adding the instantiated heads to N (line 9). The consequences of EGDs are accumulated in the term mapping μ (line 13) so, in line 14, $\mu(v)$ is defined for each labeled null v that must be replaced with $\mu(v)$. Line 13 ensures that μ is correctly updated even if several EGDs are applied in a single iteration. Since both EGDs and TGDs can be applied in an iteration, both I and N must be updated at the iteration end (lines 14–16). When $h(x_i)$ and $h(x_j)$ are both labeled nulls in line 10, we can replace either value with the other. To obtain a deterministic algorithm, we can totally order all values so that all constant values are smaller than all labeled nulls, and then always replace the larger value with the smaller one (line 12). Constant values are thus never replaced with labeled nulls, and this can also ensure uniqueness of the chase (see below).

Thus, enumerating (active) triggers, preventing repetition of derivations, and efficiently applying the chase steps lie at the core of any chase system. In the rest of this section we discuss several possibilities for realising these key components, each leading to a distinct chase variant.

The variant given in Algorithm 1 is called the *restricted chase* to stipulate that triggers are restricted only to the active ones (cf. line 6). A drawback of the restricted chase is that the chase solution is not unique (even up to isomorphism of labeled nulls), as shown in Example 1.

EXAMPLE 1. *Let Σ and and I be as in* (3) *and* (4).

$$R(x_1, x_2) \to \exists y\, R(x_1, y) \wedge A(y) \wedge A(x_2) \tag{3}$$

$$I = \{\, R(a,b), R(b,b) \,\} \tag{4}$$

Set Σ is weakly acyclic [13] so the restricted chase terminates on all finite instances, but the solution depends on the ordering of chase steps. Triggers $h_1 = \{x_1 \mapsto a, x_2 \mapsto b\}$ and

$h_2 = \{x_1 \mapsto b, x_2 \mapsto b\}$ *for* (3) *are both active in I. Applying the TGD step to* h_1 *makes* h_2 *inactive by deriving* $R(a,v_1)$, $A(v_1)$, *and* $A(b)$. *Alternatively, applying the TGD step to* h_2 *makes* h_1 *inactive by deriving* $R(b,v_1)$, $A(v_1)$, *and* $A(b)$.

The chase can be optimized by *normalizing* TGDs: for each TGD $\tau \in \Sigma$ of the form (1) where $\rho(\vec{x},\vec{y})$ can be rewritten as $\rho_1(\vec{x}_1,\vec{y}_1) \wedge \rho_2(\vec{x}_2,\vec{y}_2)$ so that $\vec{y_1} \cap \vec{y_2} = \emptyset$, we replace τ in Σ with $\forall \vec{x}\, \lambda(\vec{x}) \rightarrow \exists \vec{y}_i\, \rho_i(\vec{x}_i,\vec{y}_i)$ for $i \in \{1,2\}$. Example 2 shows how normalization can lead to smaller instances.

EXAMPLE 2. *Normalizing TGD* (3) *produces* (5) *and* (6). *Now by applying* (6) *to I from Example 1 we derive* $A(b)$, *which makes triggers* h_1 *and* h_2 *for* (5) *both inactive.*

$$R(x_1,x_2) \rightarrow \exists y\, R(x_1,y) \wedge A(y) \quad (5)$$

$$R(x_1,x_2) \rightarrow A(x_2) \quad (6)$$

Checking whether trigger h is active (line 6) can be difficult in practice, particularly for a test against $\mu(I \cup N)$; we discuss these issues in detail in Section 7. Furthermore, the dependence on the ordering of chase steps can make the chase more difficult to analyze from a theoretical point of view. This motivates several chase variants in which the check in line 6 is either eliminated or weakened.

The *unrestricted* (or *oblivious*) *chase* eliminates line 6. Such a simple solution removes the overhead of checking active triggers and the dependence on the ordering of chase steps. But, as Example 3 shows, unrestricted chase does not necessarily terminate even for weakly acyclic TGDs.

EXAMPLE 3. *For* Σ *and I defined as in Example 1, the unrestricted chase derives the following infinite set of facts.*

$$R(a,v_1),\ A(v_1),\ R(a,v_2),\ A(v_2),\ \ldots$$
$$A(b), R(b,w_1), A(w_1), R(b,w_2), A(w_2), \ldots$$

The *unrestricted Skolem chase* [26, 38, 28] also eliminates line 6, but it also skolemizes TGDs: in each TGD τ of the form (1), each existentially quantified variable $y \in \vec{y}$ is replaced with a function term $f(\vec{z})$ where f is a fresh function symbol and $\vec{z}$ contains all variables occurring in both the head and the body of τ. The chase then proceeds as in Algorithm 1, but without line 6 and by using the preprocessed τ in line 7. The result of the Skolem chase is unique (if the EGD steps are determinized as in line 8). Although cases exist where the restricted chase terminates but the Skolem chase does not, the known acyclicity conditions [18] ensure termination of both. Example 4 illustrates the Skolem chase.

EXAMPLE 4. *Normalizing and then skolemizing TGD* (3) *produces* (7) *and* (8).

$$R(x_1,x_2) \rightarrow R(x_1,f(x_1)) \wedge A(f(x_1)) \quad (7)$$

$$R(x_1,x_2) \rightarrow A(x_2) \quad (8)$$

Applying TGDs (7) *and* (8) *to I from Example 1 produces* $R(a,f(a))$, $A(f(a))$, $A(b)$, $R(b,f(b))$, *and* $A(f(b))$, *after which the chase terminates: functional term* $f(x_1)$ *in* (7) *captures the fact that the fresh null depends only on* x_1, *and so applying* (7) *to* $R(a,f(a))$ *and* $R(b,f(b))$ *does not introduce more nulls as in Example 3. Normalization is very important since it eliminates variables within Skolem terms; for example, skolemizing* (3) *directly produces* (9), *and applying* (9) *to I* does not *terminate.*

$$R(x_1,x_2) \rightarrow R(x_1,f(x_1,x_2)) \wedge A(f(x_1,x_2)) \wedge A(x_2) \quad (9)$$

Since functional terms provide canonical "names" for labeled nulls, a global counter of labeled nulls is not required, which may simplify implementation. For example, deriving the first atom of (7) can be implemented using the SQL query (10), which does not interact with other TGDs.

```
INSERT INTO R(a,b)
SELECT DISTINCT R.a, append('_Sk_f(',R.a,')') FROM R      (10)
```

The *parallel chase* [11] weakens line 6 to check whether h is active in I, rather than in $\mu(I \cup N)$; since I is fixed in an iteration, this can make checking active triggers much easier to implement. Known acyclicity conditions [18] ensure termination of the parallel chase, and the solution is deterministic, although it may be larger than the one produced by the restricted chase. Example 5 illustrates the parallel chase.

EXAMPLE 5. *Let* Σ, *I*, h_1, *and* h_2 *be as in Example 1. Both* h_1 *and* h_2 *are active for I, so the parallel chase applies the TGD step to both triggers independently and derives* $R(a,v_1)$, $A(v_1)$, $A(b)$, $R(a,w_1)$, *and* $A(w_1)$. *No active triggers exist after this step so the parallel chase terminates.*

The *single-TGD-parallel* (or *1-parallel*) *chase* checks active triggers w.r.t. all facts derived thus far apart from the ones derived by the TGD τ currently considered in line 4. As with the parallel chase, implementing this variant can be much easier than for the restricted chase (see Section 7).

Although the check in line 6 is not needed with skolemized TGDs, it can still be used, in which case we obtain the *restricted* (or *parallel* or *1-parallel*) *Skolem chase*: each of these variants never produces more facts than original variant, and it can sometimes produce fewer facts.

The *core chase* [11] extends the parallel chase by replacing after each iteration the instance with its *core* [14]—the smallest subset of I that is homomorphically equivalent to I. The core chase produces a smallest finite solution whenever one exists, but efficient computation of the core on instances of nontrivial sizes is an open problem.

4. THE SYSTEMS TESTED

As we explained in Section 1, our main goals are to determine whether available chase implementations can support data exchange and query answering on nontrivial inputs, and to investigate the implementation decisions most relevant to performance. To answer these questions, we used nine publicly available systems, which we summarize in Table 1. We group the systems based on their primary motivation.

Systems motivated by data exchange. DEMo [34] was one of the first data exchange engines. It implements the restricted chase for s-t TGDs, and target TGDs and EGDs; moreover, upon termination, it computes the core of the solution using the algorithm by Gottlob and Nash [17]. The system runs on top of an RDBMS (PostgreSQL or HSQLDB).

CHASEFUN [10] is a more recent data exchange system. It supports only s-t TGDs and functional dependencies, and it implements a variant of the unrestricted Skolem chase in

System	s-t TGDs	t TGDs	EGDs	Cert. Ans.	Engine	Strategy	Sources
Explicit chase implementations							
ChaseFun	✓		FDs only		RDBMS	unrestricted Skolem chase	
DEMo	✓	✓	✓		RDBMS	restricted chase + core computation	
Graal	✓	✓		✓	RAM	restricted chase	✓
Llunatic	✓	✓	✓	✓	RDBMS	restr./unrestr./1-parallel Skolem/fresh-nulls chase	✓
PDQ	✓	✓	✓	✓	RDBMS	restricted chase	✓
Pegasus	✓	✓	✓		RAM	restricted chase	✓
Chase-related systems							
DLV	✓	✓		✓	RAM	unrestricted Skolem chase	
E	✓	✓	✓	✓	RAM	paramodulation	✓
RDFox	✓	✓	✓	✓	RAM	restricted/unrestricted Skolem chase	✓

Table 1: Summary of the tested systems

which TGD and EGD chase steps are ordered to reduce the size of the intermediate chase results. We used an implementation that runs on top of PostgreSQL.

Systems motivated by data cleaning. LLUNATIC [16] was initially developed for data cleaning, but has since been redesigned as an open-source data exchange system that supports s-t TGDs, target TGDs and EGDs, and computing certain query answers. It implements the 1-parallel Skolem chase (the default), the unrestricted and restricted Skolem chase, and the restricted chase with fresh nulls (i.e., without Skolem terms). The system runs on top of PostgreSQL.

Systems motivated by query reformulation. We identified two systems that use the chase for query reformulation.

PEGASUS [29] is a system for finding minimal queries equivalent to a given query with respect to a set of TGDs and EGDs. It uses the Chase & Backchase method of Deutsch, Popa, and Tannen [12], which is supported using a RAM-based implementation of the restricted chase. It is not actively developed, but is available in open source.

PDQ [8, 9] takes a query, a set of integrity constraints, and a set of interface descriptions (e.g., views or access methods), and it produces an equivalent query that refers only to the interfaces and whose cost is minimal according to a preselected cost function. By extending the Chase & Backchase method, the system reduces the query reformulation problem to the problem of checking query containment under TGDs and EGDs, which is solved using an implementation of the restricted chase on top of an RDBMS.

Systems motivated by query answering. GRAAL [6] is an open-source toolkit mainly developed for computing certain answers to queries under dependencies. Although the system was not originally designed for chase computation, it uses a “saturation algorithm” that can be seen as variant of the standard chase: it applies rules to the data in a breadth-first, forward-chaining manner. GRAAL can be used both in an RAM and on secondary storage, where the latter can be provided by an RDBMS, triple store, or a graph database. We found the RAM-based version to be the most efficient, so we used that version in our experiments.

Chase-related systems. A prominent goal of our work was to investigate how chase implementations fare against systems from other communities that either (i) implement algorithms related to the chase, or (ii) can answer queries over dependencies using completely different approaches. Many systems satisfy these requirements, so we decided to restrict our attention to several prominent representatives. In particular, we considered Datalog engines in the former, and a resolution-based theorem prover in the latter category.

DLV [23] is a mature disjunctive Datalog engine that supports a range of features such nonmonotonic negation, aggregates, and user-defined functions. The system comes in several flavors: a RAM-based version that supports function symbols in the rules, another RAM-based version with native support for TGDs [22], and an RDBMS-based version that supports neither function symbols nor TGDs. The latter is not applicable to our setting, and we used the version with function symbols since it proved to be more stable. We implemented a preprocessing skolemization step, allowing DLV to support the unrestricted Skolem chase for TGDs; however, the system does not support EGDs [33]. The system can be used without a query, in which case it computes and outputs the chase solution. If a query is provided, the system evaluates the query over the chase and outputs the result. To identify only certain answers, we externally post-process the query output to remove functional terms.

RDFOX [30] is a high-performance RAM-based Datalog engine. It was originally designed for Datalog with EGDs without the unique name assumption on the RDF data model. To support the chase, RDFOX was extended as follows. First, a builtin function was added that produces a labeled null unique for the function's arguments, which emulates Skolem terms. Second, a builtin function was added that checks whether a CQ is not satisfied in the data, thus enabling both the restricted and the unrestricted chase variants. Third, a mode was implemented that handles EGDs under unique name assumption. Fourth, to support relations of arbitrary arity, a preprocessor was implemented that shreds n-tuples into RDF triples and rewrites all dependencies accordingly.

E [36] is a state of the art first-order theorem prover that has won numerous competitions. It takes as input a set of axioms $\mathscr{F}$ and a conjecture H, and it decides the unsatisfiability of $\bigwedge \mathscr{F} \wedge \neg H$. E implements the paramodulation with selection [31] calculus, of which the unrestricted Skolem chase is an instance: each inference of the Skolem chase is an inference of paramodulation (but not vice versa). Paramodulation requires $\mathscr{F}$ to be represented as a set of clauses—that is, first-order implications without existential quantifiers but possibly containing function symbols and the equality predicate. Thus, $\mathscr{F}$ can capture EGDs and the result of preprocessing TGDs as described in Section 3. Moreover, E can also be used in a mode where $\mathscr{F}$ contains arbitrary first-order formulas, thus capturing TGDs directly without any preprocessing; however, this approach proved less efficient, so we did not use it in our experiments. Finally, conjecture

H can contain free variables, in which case E outputs each substitution σ that makes $\bigwedge \mathscr{F} \wedge \neg\sigma(H)$ unsatisfiable; thus, E is interesting as it can answer queries without computing the chase in full. Ordered paramodulation targets first-order logic, which is undecidable; hence, E is not guaranteed to terminate on all inputs, not even if $\mathscr{F}$ encodes dependencies on which the unrestricted Skolem chase terminates. The system's behavior can be configured using many parameters, and we used the values suggested by the system's author.

5. THE BENCHMARK

Our CHASEBENCH benchmark allows effective testing of the correctness and the performance of chase implementations, and it consists of two distinct parts. The first part is described in detail in Section 5.1 and it comprises several tools that allow the generation and processing of test data in a common format. The second part is described in detail in Section 5.2 and it comprises a number of *test scenarios*, each consisting of a (i) schema description, (ii) a source instance, (iii) sets of s-t TGDs, target TGDs, and/or target EGDs, and (iv) possibly a set of queries. We used existing resources whenever possible; for example, we repurposed scenarios produced by the IBENCH metadata generation tool [4], as well as the instances of varying sizes produced by the instance-generation tool TOXGENE [7]. We divide our scenarios into the following two groups.

Correctness scenarios were designed to verify that the systems correctly produce universal solutions. Since checking homomorphisms between solutions is computationally challenging, these scenarios were developed to generate very small solutions. Moreover, these scenarios do not contain queries, as correctness of query answering can be verified by simply comparing the certain answers on larger scenarios.

Data exchange and query answering scenarios aim to test the performance of computing the target instance and of answering queries over the target instance in data exchange. These scenarios vary the data size and the complexity of the dependencies to simulate different workloads.

5.1 Test Infrastructure

1. The common format. In their "native" versions, the systems from Section 4 take a wide range of input formats. For example, RDBMS-based systems expect the source instance to be preloaded into a relational database, whereas RAM-based systems typically read their input from files. Moreover, the structure of the inputs varies considerably; for example, RDFOX expects data to be represented as triples, and E expects data to be encoded as first-order sentences in the TPTP format. The translation between various input formats is often straightforward, but in certain cases (e.g., if schema shredding or the transformation of TGDs is required) it can considerably affect a system's performance.

To allow for an automated and fair comparison of all systems regardless of their implementation details, we standardized the input structure for all systems. Theorem-proving formats such as TPTP can express first-order sentences and can thus represent dependencies and queries, but we found these inappropriate for CHASEBENCH as they are difficult to read for humans. We thus developed our own "common format" for describing all parts of a scenario (i.e., the schema description, the source instance, the dependencies, and the queries). We also wrote a parser for the common format, which was used to develop wrappers for the systems. The wrapper was provided by the system designers whenever possible, but for E, DLV, and PEGASUS we developed the wrappers ourselves. Our tests required each system to read the scenarios in the common format, so our test results cover all times necessary to transform the inputs.

2. The instance repair tool. Generating scenarios with EGDs and large source instances is complex since, due to the size of the source instance, it is difficult to ensure that no EGD chase step fails. For example, TOXGENE does not necessarily generate instances that are consistent with a collection of TGDs and EGDs, and in fact the chase failed on all instances that the system produced initially so this problem is not merely hypothetical. Thus, we developed a simple instance repair tool. Given a set of dependencies Σ and an instance I on which the chase of Σ fails, the tool proceeds as follows. First, the tool computes the chase of Σ and I without the unique name assumption: when equating two distinct constant values, one value is selected as *representative* and the other one as *conflicting*, the latter is replaced with the former, and the chase continues. Second, the tool removes from I all facts containing a conflicting value encountered in the previous step. The chase of Σ and the subset of I produced in this way is guaranteed to succeed. This strategy proved very effective in practice: on average it removed slightly more than 1% of the facts from I, and so the size and the distribution of the source instance were largely unaffected.

3. The target TGD generator. In addition to generating large source instances, generating scenarios with a significant number of s-t TGDs, target TGDs, and target EGDs was critical for adequately evaluating the performance of the tested systems. One of our goals was to push the systems to their limit by testing them on *deep chase scenarios* that generate very large instances using long chase sequences. To this end, we developed a custom target TGD generator that can generate weakly acyclic TGDs while controlling their depth and complexity. The generator is based on our earlier work [20], and it is described in more detail in Appendix A. In our experiments we used the generator to develop scenarios from scratch, but the tool can also be used to increase the difficulty of existing scenarios.

4. The query generator. To adequately evaluate the performance of computing certain answers, nontrivial queries over the target schema are required. Existing benchmarks such as LUBM [19] come with several manually curated queries, which we incorporated into our scenarios. In order to obtain queries for scenarios where the target schema is automatically generated, we developed a new query generator. The generator allows controlling the number of joins in the query and, importantly, it ensures that the output of the generated queries on the target instance is not empty. The generator is described in more detail in Appendix B.

5. The homomorphism checker. Checking correctness of computing certain query answers is easy: certain answers are unique for the query so the systems' outputs can be compared syntactically. Checking correctness of the chase is more involved since the result of the restricted chase is not unique and, even with the unrestricted Skolem chase, it is

unique only up to the renaming of Skolem functions. Hence, to verify the correctness of the systems, we developed a tool that can check the existence of homomorphisms, mutual homomorphisms, and isomorphisms between instances. The tool enumerates all candidate homomorphisms using brute force, so it can be used only on relatively small instances (i.e., few thousands facts). Consequently, we designed our correctness scenarios so that they produce small solutions.

5.2 Test Scenarios

Our benchmark consists of a total of 23 scenarios, each comprising a source and target schema, a source instance, a set of dependencies, and possibly a set of queries; all dependencies in all scenarios are weakly acyclic. We classify the scenarios in five families, shown in Table 2. The first family contains six small scenarios for testing correctness of data exchange, whereas all other families are aimed at testing the performance of computing the target instance and the certain answers. The IBENCH and the LUBM families were derived from the well established benchmarks in the database and the Semantic Web communities, respectively. Finally, we developed the MANUALLY CURATED and the DEEP families ourselves to test specific aspects of the chase. We discuss next the main features of these families. We identify scenarios using an identifier that, in most cases, combines the family identifier with the source instance size; for example, DOCTORS-10k is a scenario with 10 k source facts.

a. Correctness tests. As we explained in Section 5.1, our homomorphism checker can handle only small instances. We thus prepared six scenarios that produce small chase results, while aiming to cover exhaustively the different combinations of various features. All scenarios contain s-t TGDs and test different combinations such as joins over the source schema, vertical partitioning, and self-joins both in the source and in the target schemas. The scenarios cover standard examples from some of the prior papers and surveys on the chase (e.g., [32, 27]), including cases where TGDs and EGDs interact, where the chase fails, and where various acyclicity conditions are used to ensure chase termination.

b. Manually curated scenarios. Our manually curated scenarios are based on the DOCTORS data integration task from the schema mapping literature [15]. These scenarios are relatively small in terms of the number of relations, attributes, and dependencies (cf. Table 2), but we believe that they represent a useful addition to the benchmark for two reasons. First, these scenarios are based on schemas inspired by real databases about medical data. Second, they simulate a common use case for data exchange: take two databases from the same domain but with different schemas and bring them to a unified target representation. We used TOXGENE to generate instances of the source schema of 10 k, 100 k, 500 k, and 1 M facts. DOCTORS contains EGDs that refer to more than one relation in the body, which cannot be handled by all systems in our evaluation. Hence, we also generated a simplified version, called DOCTORSFD, that contains only EGDs corresponding to functional dependencies. Consequently, the manually curated family contains eight scenarios. We also used the query generator described in Section 5.1 to generate nine queries covering most of the possible joins among the three target relations.

c. LUBM scenarios. LUBM [19] is a popular benchmark in the Semantic Web community. It does not simulate a data exchange task, but it is useful as it comes with nontrivial target TGDs and, more importantly, queries designed to test various aspects of query answering. Using the LUBM data generator we produced instances with 90 k, 1 M, 12 M, and 120 M facts and transformed them as follows.

- The LUBM generator produces data as RDF triples, which we converted into a relational form by vertical partitioning: a triple $\langle s,p,o\rangle$ is transformed into a unary fact $o_{src}(s)$ if $p = rdf{:}type$, and into a binary fact $p_{src}(s,o)$ of the source instance otherwise.
- For each unary relation o_{src} from the previous step, we added the s-t TGD $\forall x\; o_{src}(x) \rightarrow o(x)$, and similarly for each binary relation. Thus, the s-t TGDs simply copy the source instance into the target instance.
- The dependencies in LUBM are encoded using an ontology, which we converted into target TGDs using vertical partitioning and the known correspondences between description logics and first-order logic [5].
- We manually converted all SPARQL queries into CQs; this was possible for all queries.

As a result of these transformations, the source and the target schemas of LUBM contain only unary and binary relations. Also, the source instance of LUBM-120M is much larger than any other source instance in our scenarios.

d. IBENCH scenarios. IBENCH [4] is a tool for generating dependencies whose properties can be finely controlled using a wide range of parameters. For our purpose, we selected two existing sets of dependencies [4, Section 5] that consist of second-order TGDs, primary keys, and foreign keys. To obtain dependencies compatible with most of our systems, we modified a parameter in the IBENCH scripts to generate ordinary s-t TGDs instead of second-order TGDs. We thus obtained two scenarios of the IBENCH family. (*i*) STB-128, derived from an earlier ST-benchmark [3], is the smaller scenario of the family. (*ii*) ONT-256, a scenario motivated by ontologies, is several times larger than STB-128.

For both scenarios, we used the integration of IBENCH and TOXGENE to generate 1 k facts per source relation. Next, we used our instance repair tool from Section 5.1 to ensure that the chase does not fail. Finally, we generated 20 queries for each scenario using our query generator.

e. The DEEP scenarios. The final family of scenarios was developed as a "pure stress" test. We used our target TGD generator to generate three scenarios with 1000 source relations, 299 target relations, 1000 linear s-t TGDs, and increasing numbers (100, 200, and 300) of linear target TGDs. Moreover, to generate the source instance, we globally fixed a substitution σ that maps each variable $x \in \mathsf{Vars}$ into a distinct constant value $\sigma(x) \in \mathsf{Const}$; then, for each linear s-t TGD with $R(\vec{x})$ in the body, we added $\sigma(R(\vec{v}))$ to the source instance. Thus, all source instances contain just one fact per relation; however, the TGDs are very complex so they produce over 500 M facts on the largest DEEP300 scenario.

The TGDs were taken from our previous work [20] and they have the following structure. All TGD heads have three relations joined in a chain. Each atom has arity four and each dependency can have up to three repeated relations. The three head predicates and the body predicate have been

Scenario		Source Schema		Target Schema		s-t	t TGDs		EGDs		Qrs	Source Instance Facts
Family	Name	Rel	Attr	Rel	Attr	TGDs	Tot	Inc.Dep	Tot	FDs		
Corr.	EmpDept	1	3	2	5	1	2	2	0	0	0	1
Corr.	Tgds-a	1	3	5	12	2	5	5	0	0	0	1
Corr.	Tgds-b	2	8	3	9	5	2	1	0	0	0	8
Corr.	Egds	1	2	1	2	1	0	0	1	1	0	3
Corr.	TgdsEgds-a	1	3	5	12	3	5	5	4	4	0	4
Corr.	TgdsEgds-b	1	3	5	12	6	5	5	4	4	0	80
Man.C.	DoctorsFD	3	24	5	17	5	0	0	8	8	9	10 k, 100 k, 500 k, 1 M
Man.C.	Doctors	3	24	5	17	5	0	0	10	8	9	10 k, 100 k, 500 k, 1 M
LUBM	LUBM	30	76	74	179	30	106	91	0	0	14	90 k, 1 M, 12 M, 120 M
iBench	STB-128	111	535	176	832	128	39	39	193	193	20	150 k
iBench	Ont-256	218	1210	444	1952	256	273	273	921	921	20	1 M
Deep	Deep100	1000	5000	299	1495	1000	100	50	0	0	20	1 k
Deep	Deep200	1000	5000	299	1495	1000	200	100	0	0	20	1 k
Deep	Deep300	1000	5000	299	1495	1000	300	150	0	0	20	1 k

Table 2: Summary of the test scenarios

chosen randomly out of a space of 300 predicates. We generated 10% of all TGDs from a smaller subset of predicates of size 60. Also, around 10% of the s-t TGD heads were constructed by getting the body and two (out of three) head atoms of a target TGD. This causes some target TGDs to almost map entirely to these particular s-t TGDs. After generating each dependency, a weak acyclicity test was used to discard the dependency if acyclicity was violated.

6. SYSTEM COMPARISON

We ran a total of 40 tests per system: 6 correctness tests, 17 chase computation tests, and 17 query answering tests. Our correctness tests revealed a number of errors in the systems, most of which were corrected by the system authors during the evaluation period. Pegasus failed two correctness tests but could not be updated as it is not under active development; all other systems eventually passed all correctness tests. Complete results of the performance tests, including a breakdown of all times, are given in Appendix C and on the benchmark Web site. Figure 1 summarizes some results that we discuss next.

Hardware configuration. We ran all tests on a server with six physical 1.9 GHz Xeon v3 cores, 16 GB of RAM, and a 512 GB SSD, running Ubuntu v16. Our configuration is thus not very far from that of a high-end laptop.

Test Setup. All systems apart from E and DLV (as we discuss shortly) were required to perform the following steps:

1. load the source instance from the .csv files on disk;
2. load the dependencies in the common format;
3. run the chase of the s-t TGDs;
4. run the chase of the target dependencies;
5. save the target instance to .csv files on disk; and
6. run each query and save its results to .csv files on disk.

For each scenario, each system was allowed three hours to complete; if the system ran out of time or memory, we count the scenario in question as failure and report no results (not even the loading times). In order to analyze the relative contribution of all the steps, we captured the times for all steps independently whenever the systems allowed that. We also repeated the experiments with s-t TGDs only.

Figure 1 shows (I) the *chase execution times* (steps $3+4$), (II) the *source import and target export times* (steps $1+5$), (III) the *s-t TGDs chase times* (step 3), (IV) the total chase times (steps $1+2+3+4$), and (V) the *query execution times* (step 6) for the scenarios and where the results vary significantly between different system configurations. Figure 1 shows the results only for the 1-parallel Skolem chase for Llunatic and the unrestricted Skolem chase for RDFox. All results for all scenarios and all chase variants supported by the systems are given in Appendix C. The data sizes in the STB-128 and Ont-256 scenarios do not vary so we report our results as a single bar chart; for all other scenarios we use line charts that show scaling with the data size. Test coverage of different systems is reported separately.

E and DLV report no intermediate times so we treated them differently. The reasoning strategy of DLV is closely related to the Skolem chase so we report its "total chase time"; however, we found no reasonable analogous time for E. Moreover, to compare E and DLV with the other systems, in Figure 1.VI we show the query evaluation times measured in a different way: (i) for DLV and E, we report the total time needed to answer all queries; and (ii) for all other systems we report the import and chase times multiplied by the number of queries (thus compensating for the fact that, for each query, DLV and E load the dataset and perform reasoning from scratch), plus the sum of all query times.

A "db-to-db" protocol, where the source and the target instances are stored in an RDBMS, might have been more natural for RDBMS-based systems. Our "file-to-file" approach, however, has two advantages. First, even in RDBMS-based systems importing the data may require indexing or building supporting data structures (e.g., we discuss the dictionary encoding in Section 7), which can incur overheads; thus, reporting the import and the chase times separately allows us to investigate these issues. Second, reporting the cumulative times allows us to take DLV and E into account.

Test coverage. On some tests, certain systems did not support all required feature, they terminated with an exception, or did not finish in three hours; such tests are not shown in Figures 1. Table 3 summarizes test coverage for all the systems. For each system and each of the three test categories, we report the number of tests that the system was applicable to and the number of successfully completed tests; furthermore, we report the causes of failures (if any). Note that for most systems a failure in data exchange on a test scenario implies failure in the corresponding query answering tests,

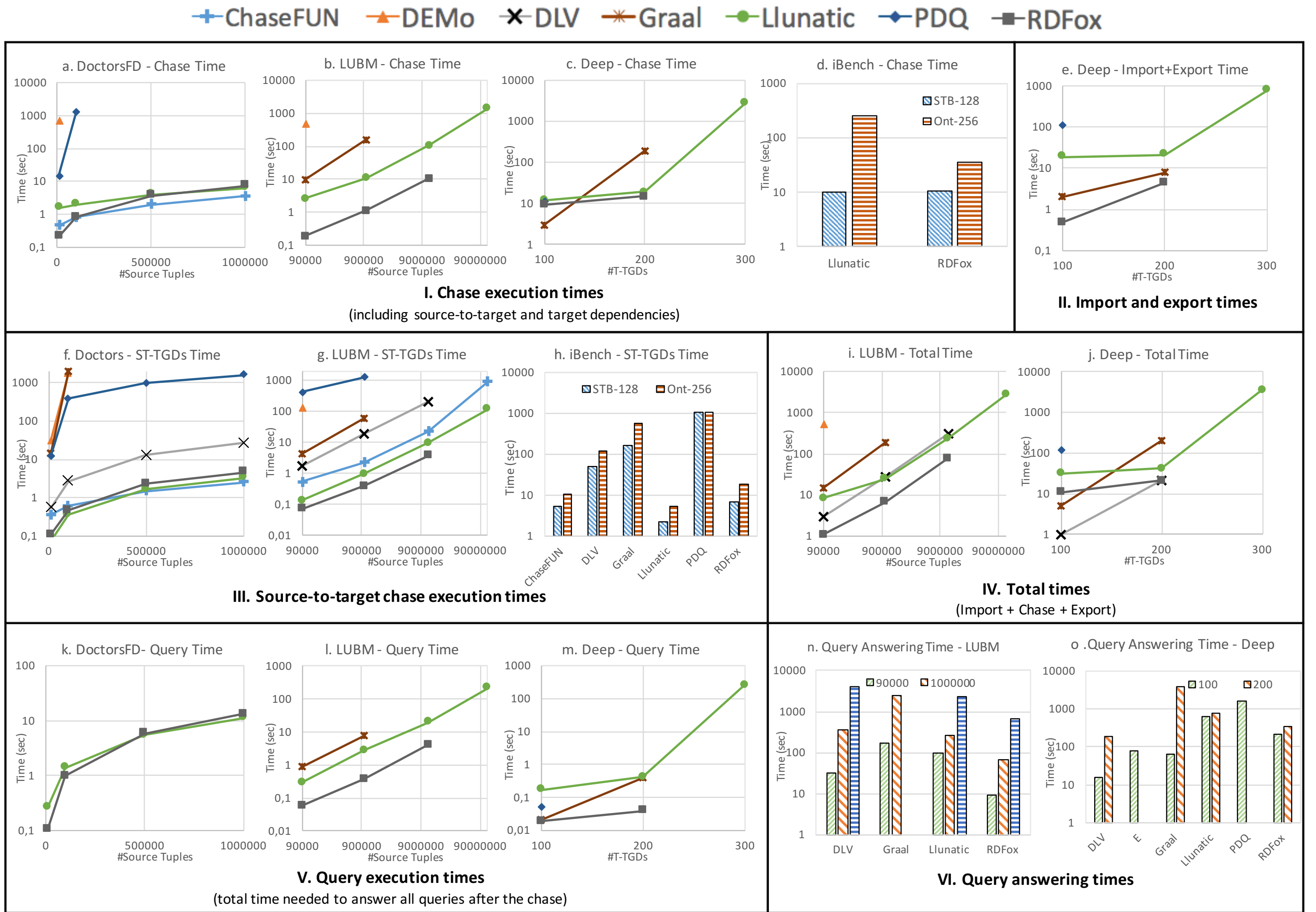

Figure 1: Experimental results for scenarios where times vary significantly between system configurations

and we count such failures only once in the table.

As one can see, LLUNATIC was the only system that completed all tests, closely followed by RDFOX which ran out of memory on LUBM-120M and DEEP300. DEMO and the systems that use chase for query reformulation exhibited low coverage; for example, PDQ completed only five out of the 17 data exchange tests. CHASEFUN computed the chase in all tests that it was applicable to.

7. LESSONS LEARNED

Our benchmark should not be misunderstood as aiming primarily for a performance competition: the tested systems were designed under completely different assumptions, often targeting tasks currently not covered by our benchmark. A key feature of our benchmark is that it covers a range of scenarios, which allowed us to answer the questions we posed in Section 1. Specifically, we could observe the behavior of the chase on a diverse set of systems and inputs, which allowed us to derive many general conclusions about the tradeoffs in implementing different chase variants (see Section 3). Moreover, we could compare the effectiveness of the systems specifically designed for the chase with that of the systems that tackle related problems. We summarize our findings in the rest of this section.

Restricted vs. unrestricted vs. parallel chase. A major decision facing a chase system designer is whether to implement the active trigger check (line 6 of Algorithm 1). By analyzing the performance of different chase variants implemented in LLUNATIC and RDFOX on our scenarios (Table 4 shows some, and Table 7 in Appendix C shows all results), we obtained several important insights. Interestingly, the tradeoffs associated with this question are quite different for RDBMS- and RAM-based systems.

Implementing the restricted chase in RDBMS-based systems is quite challenging. Such systems retrieve triggers using SQL queries, so it is natural to embed the active trigger check into the queries. For example, query (11) attempts to retrieve the active triggers for TGD (3) from Example 1.

```
SELECT DISTINCT R.a FROM R WHERE NOT EXISTS
  (SELECT * FROM R AS R1, A AS A1, A AS A2        (11)
    WHERE R.a=R1.a AND R1.b=A1.a AND R.b=A2.a)
```

RDBMSs, however, evaluate queries fully before any updates, so query (11) cannot detect that applying a chase step to one trigger makes another trigger inactive (as in Example 1). To properly implement the restricted chase, we must check triggers one-by-one using independent queries; for example, we can add `LIMIT 1` to (11) and run one query per trigger, but this is very slow on nontrivial datasets in most RDBMSs. Table 7 confirms this: with both the 1-parallel

System	Tests Run				Tests Completed				Failures	
	Corr.	Chase	Query	Total	Corr.	Chase	Query	Total	Timeouts	Memory
Explicit chase implementations										
CHASEFUN	1	4	0	**5**	1	4	0	**5**	0	0
DEMO	6	17	0	**23**	6	3	0	**9**	11	3
GRAAL	3	7	7	**17**	3	4	4	**11**	0	3
LLUNATIC	6	17	17	**40**	6	17	17	**40**	0	0
PDQ	6	17	17	**40**	6	5	5	**16**	12	0
PEGASUS	6	17	0	**23**	4	0	0	**4**	17	0
Chase-related systems										
DLV	3	7	7	**17**	3	5	5	**13**	2	0
E	6	0	17	**23**	6	0	3	**9**	14	0
RDFOX	6	17	17	**40**	6	15	15	**36**	0	2

Table 3: Test coverage

	LUBM-90k			DEEP200		
	Ch.Time	# Facts	Query T.	Ch.Time	# Facts	Query T.
LLUNATIC 1-Parallel	2.67	141,213	0.29	19.25	902,636	0.41
LLUNATIC Unrest	6.37	177,738	0.31	33.35	926,324	0.44
LLUNATIC Rest	2196.00	141,213	0.23	7521.00	893,990	0.36
RDFOX Unrest	0.19	177,738	0.06	15.23	926,324	0.03
RDFOX Rest	0.21	141,213	0.06	24.02	892,516	0.03

Table 4: Results for variants of the chase

and the unrestricted Skolem chase, LLUNATIC runs orders of magnitude faster than with the restricted Skolem chase, and some of the latter tests timed out.

Query (11), however, can be used to implement the 1-parallel chase, which can even eliminate a common source of overhead: by combining `DISTINCT` and the active triggers check, the query never produces duplicate answers, so separate duplicate elimination is not needed. Indeed, as Table 7 shows, LLUNATIC is faster with the 1-parallel Skolem chase than with the unrestricted Skolem chase.

In contrast, RAM-based systems can more efficiently interleave queries with updates, which can make the active triggers check easier. For example, RDFOX identifies active triggers using a subquery similar to (11), but its optimized RAM-based indexes [30] can efficiently answer the `NOT EXISTS` subqueries while taking into account the result of all concurrent updates. Thus, as Table 7 shows, the performance of the restricted and the unrestricted Skolem chase in RDFOX differs by only a couple of seconds.

Solution sizes and getting to the core. Another question to consider is the impact of the chase variant on the size of the universal solution. Our benchmark again allowed us to investigate this issue: Table 4 shows the solution sizes obtained by the chase variants in LLUNATIC and RDFOX (Table 7 in Appendix C shows all results). As one can see, solutions produced by the restricted chase can be between 4% and 21% smaller than those produced by the unrestricted chase; however, we did not observe any significant impact on the performance of query answering. Thus, we conclude that the choice of the chase variant can be mainly driven by the ease of implementation, rather than the solution size.

An interesting question is whether computing the core can further reduce solution sizes. To this end, we ran DEMO to compute the core of the universal solutions for scenarios in our benchmark. DEMO computed the core for DOCTORS-10k and LUBM-90k; the computation failed on all other scenarios, and the core was in both cases of the same size as the result of the restricted chase. We were not able to test the impact of computing the core on most of our scenarios: as we discuss in Section 8, computing the core of large instances is an important open problem. We could, however, answer this question partially: a set Σ of s-t TGDs can be rewritten into a set Σ' such that the restricted chase of Σ' returns the core of the universal solution for Σ [28]. We ran this experiment on the DOCTORS scenario and only s-t TGDs; full results are shown in Table 8 in Appendix C. On the 10k and the 100k scenarios, the core target instances were 18% smaller then the ones produced using unrestricted Skolem chase, suggesting that methods for computing the core in a more general setting could be practically relevant.

Implementing EGDs. To propagate the consequences of EGDs in lines 14 and 16 of Algorithm 1, a system will typically first retrieve and delete all affected facts, apply μ, and insert the result back into the instance. These operations require mass updates that are much more expensive in an RDBMS-based than a RAM-based system, which makes supporting EGDs in an RDBMS very challenging. Our benchmark results offer evidence for this observation: the chase times of LLUNATIC were considerably higher than of RDFOX on the ONT-256 scenario, which had the largest and most complex set of EGDs (see Figure 1.I.d).

Representing labeled nulls. Chase systems must consistently distinguish labeled nulls from constant values, which turned out to be a source of complexity in RDBMS-based systems. We noticed two common solutions to this problem.

RDBMS-based systems represent labeled nulls using a suitable encoding, often string-based; for example, string values are encoded in query (10) using the `'_Sk_'` prefix. Each attribute is thus represented using just one relation column, so join conditions are expressed naturally. Nevertheless, value decoding requires pattern matching; for example, to compute certain answers, LLUNATIC filters string-typed labeled nulls using `NOT LIKE '_Sk_'` in the `WHERE` clause. This is a considerable source of overhead, for two reasons. First, pattern matching is costly. Second, this kind of conditions are not handled nicely by the query planner: as usual, the

optimizer pushes selections down towards the scans, but this forces the database engine to apply pattern matching to most facts in the target relations. In most cases, filtering out labeled nulls as a last step of the query would be faster, since the number of facts to analyze is much smaller. This effect can be observed in the query times of LLUNATIC on DEEP300, one of our very large scenarios: the system answers Query 16 in 147 s, but of these, 118 s (80%) are used to filter out labeled null; similarly, Query 20 takes 44 s, of which 30 s (68%) are used to filter out labeled nulls.

There are no obvious solutions to this issue. One could implement an alternative query execution strategy that forces the query planner to materialize the intermediate result of the query and then filter labeled nulls at the very end, but this incurs a materialization overhead. An alternative is to adopt a multi-column representation of labeled nulls. For example, DEMO represents labeled nulls using three columns per target relation attribute: one boolean column determines whether the attribute contains a constant value or a labeled null, one column stores constant values, and one column stores labels of labeled nulls. This cleanly separates labeled nulls from constant values; however, it triples the size of the database, and it greatly complicates join conditions, which also often confuses the query optimizer.

In summary, our benchmark allowed us to examine the drawbacks of both approaches, to the point that we consider the development of new, performance-oriented representations of labeled nulls an interesting open research question.

Query execution and query characteristics. All systems that successfully computed the chase also successfully evaluated the appropriate queries. Most queries can be answered very quickly (typically under 1 s). Queries were slowest on DEEP300 and LUBM-120M because the target instances were much larger than in other scenarios.

We analyzed the queries in our benchmark to detect a possible correlation between execution times and different complexity parameters, such as the number of variables in the head and the body, the number of joins and so on; Table 9 in Appendix C shows the parameters for all queries. We observed no clear correlation between the query parameters and the query answering times. Moreover, we found removing duplicates and filtering out labeled nulls to be significant sources of overhead for query answering, and these have no clear connection to the shape and size of the query.

Dictionary encoding. Columnar databases often compress data using a *dictionary encoding*, which can be applied in the chase setting as follows:

- one fixes an invertible mapping e of values to integers;
- the set of dependencies Σ and the input instance I are encoded as $\Sigma_e = e(\Sigma)$ and $I_e = e(I)$, respectively;
- the encoded chase J_e of Σ_e and I_e is computed; and
- J_e is decoded as $J = e^{-1}(J_e)$ by inverting the mapping.

Clearly, J is the chase of Σ and I. This process improves performance in a number of ways. First, the encoded instance is usually much smaller than the original. Second, comparing integers is faster than comparing strings. Third, access structures such as indexes are smaller so joins are faster. Fourth, dictionary encoding removes a problem specific to the Skolem chase: chasing target TGDs may produce very deep Skolem terms that can be expensive to manage.

DLV, GRAAL, and RDFOX all use a variant of this idea as they load the input, and E achieves similar benefits using term indexing [35]. Dictionary encoding is less common in RDBMS-based systems, but it is even more useful as it reduces the amount of data to be transferred from secondary storage, and we were again able to evaluate the impact of this optimization using the benchmark. To this end, we ran LLUNATIC with and without the encoding on LUBM-120M and DEEP300. To isolate the impact of the encoding, the source instance was available (but not encoded) in the RDBMS. In one case, we measured the chase time on the unencoded instance. In the second case, we measured the total time needed to encode the source instance, run the chase, and decode the solution. Despite the overhead of the encoding process, the execution with dictionary encoding was 46% faster on LUBM-120M and 58% faster on DEEP300. Please note that, in our experiments LLUNATIC was configured to turn on the encoding on all scenarios with target TGDs.

Chase vs. first-order theorem proving. E successfully answered all queries in less than a minute on the DOCTORS-10k and DOCTORSFD-10k scenarios, but it was not as efficient on the remaining scenarios (all times are available on the benchmark web site): it took more than an hour for DOCTORS-100k and DOCTORSFD-100k, and it ran out of memory on LUBM-1M, LUBM-12M, and LUBM-120M. Nevertheless, although E was not specifically designed for answering queries over dependencies, it could still process nontrivial scenarios. Note that the Skolem chase is actually an instance of the theorem proving calculus used in E, so the performance gap between E and the other systems is most likely due to the generality of the former.

This generality can actually be beneficial in some cases. As Figure 1.VI.o shows, E performed very well on DEEP100 by answering each query in several seconds, so we analyzed the saturated set of clauses produced by E. We noticed that, by a combination of forward and backward reasoning, the system derived many intermediate clauses ("lemmas"). Some lemmas were obtained by composing s-t TGDs with target TGDs to obtain new s-t TGDs, which introduced "shortcuts" in proofs and thus sped up query answering. In fact, queries over weakly-acyclic linear dependencies (i.e., dependencies with just one atom in the body, which covers all DEEP scenarios) can always be answered fully using such an approach [2]. Thus, E "discovered" this method using a general first-order theorem proving technique.

Query reformulation vs. query answering. Systems that use chase to support query reformulation (i.e., PEGASUS and PDQ) fared badly on all tests. The chase implementation in these systems is optimized for small instances that are obtained from the queries being reformulated, rather than for instances of sizes found in data exchange.

Maturity of the chase. Despite the increasing complexity of the test scenarios, some consisting of over 1000 dependencies and generating up to 500 M facts, several systems successfully completed most tests on mainstream hardware (a CPU with 6 cores, 16 GB of RAM, and a 512 GB SSD). In addition, some systems were able to complete the chase and answer the queries in the majority of the tests within a few minutes. Thus, our results suggest that applying the chase to tasks of nontrivial sizes is practically feasible.

8. FUTURE CHALLENGES

Our benchmark also gave some insight regarding directions for future work.

Modular implementations. While a good benchmark should provide a range of workloads for testing the chase, systems with a more modular architecture are needed in order to test hypotheses about the performance of chase. A prominent example of this kind is comparing RDBMS- and RAM-based systems: one would ideally use a system that can work either in RAM or on top of an RDBMS, and it would allow one to measure the impact of this choice independently of the myriad of other implementation factors; however, no such system exists at present. A more modular chase implementation would also be beneficial in practice since many design choices (including the one above) are appropriate for some scenarios and not for others.

Implementing labeled nulls. We showed how the implementation of labeled nulls represents an important factor of performance in query answering on an RDBMS, and that both encoding-based and multicolumn solutions have limitations. Deriving new and improved strategies to represent and manipulate labeled nulls is in our opinion an interesting and relevant open research question.

Computing the core. An open question is to what extent can computing the core reduce the size of the target instance. We investigated this for s-t TGDs, but to answer this question more generally scalable techniques for computing the core in the presence of target dependencies are needed.

Approaches to query answering. One use of the chase is for computing the certain answers to queries—a task that, in some cases, can also be tackled using radically different reasoning techniques such as theorem-provers or query-rewriting algorithms. In this paper we took a first look at comparing the chase to the other approaches, but more work is needed to further compare and possibly even combine the chase with the related approaches. For example, it would be interesting to see whether the *combined* approaches to query answering in description logics [21, 24] can be generalized to practically relevant classes of TGDs and EGDs.

Firing order. We found considerable evidence in our experiments that ordering chase steps is important in practice; for example, the good performance of CHASEFUN is due to its careful ordering of EGD and TGD steps. Still, more research is needed to understand the impact of step ordering since, even with TGDs only, a particular order can make the active triggers check more effective.

9. RELATED WORK

Experimental evaluations of many data exchange systems have already been conducted [34, 28, 27, 15, 20, 10]. In many cases we reused and extended systems from these earlier studies, but our work differs in several important ways. First, previous studies have involved a limited number of systems, typically one or two: to the best of our knowledge, this is the first attempt to compare a large number of a very different, yet related systems. Second, earlier studies have considered only a narrow range of scenarios closely related to the function of the system. We evaluate the systems on over 20 scenarios covering a wide range of assumptions, testing both correctness and scalability, covering the full spectrum of steps related to the chase from data loading to query execution. Finally, the systems, datasets, and scenarios from the earlier studies have not been made available to the community—an important step towards advancing the experimental culture of the community.

Previous efforts that are more similar in spirit to our work include ST-Benchmark [3] and iBench [4]. ST-Benchmark is concerned with evaluating tools for generating *schema mappings*. It consists of a number of mapping tasks, where each task comprises two schemas, possibly in different data models, and a description of a transformation between them. In addition, ST-Benchmark provides a generator for mapping specifications themselves. It also modifies TOXGENE—a generator of schema instances with certain properties. The tests reported in [3] focus on common transformations on a nested relational model, as opposed to relational data exchange. The ST-Benchmark suite of tools is no longer available, so we could not reuse it. However, several pieces of the infrastructure used in the ST-benchmark, such as TOXGENE and IBENCH (a successor of the specification generator from the ST-benchmark), play a prominent role in our work.

IBENCH [4] is a tool for generating metadata for benchmarking mapping scenarios. It can generate source and target schemas, s-t TGDs, and target EGDs. It is publicly available, it provides some support for data generation via TOXGENE, and it has already been used for testing schema mapping tools. In our work we complemented IBENCH with a number of additional tools, as discussed in Section 5.

10. CONCLUSIONS

Our work provides the first broad look at the performance of chase systems. Our contributions include a new benchmark that comprises test infrastructure and many test scenarios, experimental results for nine prominent systems, and insights about aspects of the systems' implementation. We intend to maintain and extend our infrastructure for benchmarking the chase presented in this paper as new systems appear and as existing systems are improved. We feel that it can be easily extended to support new classes of dependencies and other chase-related tasks, for example query reformulation with respect to dependencies.

We hope that our infrastructure will have an impact beyond the specific techniques examined here. As mentioned in the introduction, many automated reasoning communities, from SMT solving to description logics, have invested enormous effort in evaluation in the past years. In contrast, while the research literature on extending database systems with reasoning capabilities is extensive, evaluation methodologies are much less developed. A major contribution of this work, beyond its results on the chase, is as a preliminary step in addressing this gap. Evaluation infrastructure is not just a matter of providing synthetic data and dependency generators: it includes common formats, common test harnesses, and much more (see Section 5). We hope that some of our infrastructure and methods will spur activity in other evaluation tasks around reasoning in database systems.

Finally, in this work we took an initial step in comparing reasoning systems produced by the database community with the systems developed by other related communities.

11. REFERENCES

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

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

[3] B. Alexe, W.-C. Tan, and Y. Velegrakis. STBenchmark: towards a benchmark for mapping systems. In *VLDB*, 2008.

[4] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The iBench integration metadata generator. In *VLDB*, 2015.

[5] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2007.

[6] J.-F. Baget, M. Leclère, M.-L. Mugnier, S. Rocher, and C. Sipieter. Graal: A toolkit for query answering with existential rules. In *RuleML*, 2015.

[7] D. Barbosa, A. Mendelzon, and K. Keenleyside, J.and Lyons. ToXgene: A template-based data generator for XML. In *SIGMOD*, 2002.

[8] M. Benedikt, J. Leblay, and E. Tsamoura. PDQ: Proof-driven query answering over web-based data. In *VLDB*, 2014.

[9] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. In *VLDB*, 2015.

[10] A. Bonifati, I. Ileana, and M. Linardi. Functional Dependencies Unleashed for Scalable Data Exchange. In *SSDBM*, 2016.

[11] A. Deutsch, A. Nash, and J. Remmel. The chase revisited. In *PODS*, 2008.

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

[13] R. Fagin, P.G. Kolaitis, R.J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[14] R. Fagin, P.G. Kolaitis, and L. Popa. Data Exchange: Getting to the Core. *TODS*, 30(1):174–210, 2005.

[15] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, 2014.

[16] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's All Folks! LLUNATIC Goes Open Source. In *VLDB*, 2014.

[17] G. Gottlob and A. Nash. Efficient Core Computation in Data Exchange. *J. of the ACM*, 55(2):1–49, 2008.

[18] B. Cuenca Grau, I. Horrocks, M. Krötzsch, C. Kupke, D. Magka, B. Motik, and Z. Wang. Acyclicity notions for existential rules and their application to query answering in ontologies. *JAIR*, 47:741–808, 2013.

[19] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3), 2011.

[20] G. Konstantinidis and J. L. Ambite. Optimizing the chase: Scalable data integration under constraints. In *VLDB*, 2015.

[21] R. Kontchakov, C. Lutz, D. Toman, F. Wolter, and M. Zakharyaschev. The Combined Approach to Ontology-Based Data Access. In *IJCAI*, pages 2656–2661, 2011.

[22] N. Leone, M. Manna, G. Terracina, and P. Veltri. Efficiently computable Datalog$^\exists$ programs. In *KR*, 2012.

[23] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *TOCL*, 7(3):499–562, 2006.

[24] C. Lutz, D. Toman, and F. Wolter. Conjunctive Query Answering in the Description Logic $\mathscr{EL}$ Using a Relational Database System. In *IJCAI*, pages 2070–2075, 2009.

[25] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *TODS*, 4(4):455–469, 1979.

[26] B. Marnette. Generalized Schema Mappings: From Termination to Tractability. In *PODS*, 2009.

[27] B. Marnette, G. Mecca, and P. Papotti. Scalable Data Exchange with Functional Dependencies. In *VLDB*, 2010.

[28] G. Mecca, P. Papotti, and S. Raunich. Core Schema Mappings: Scalable Core Computations in Data Exchange. *Inf. Systems*, 37(7):677–711, 2012.

[29] M. Meier. The backchase revisited. *VLDB J.*, 23(3):495–516, 2014.

[30] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *AAAI*, 2014.

[31] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7. Elsevier Science, 2001.

[32] A. Onet. The chase procedure and its applications in data exchange. In *DEIS*, pages 1–37, 2013.

[33] S. Perri, F. Scarcello, G. Catalano, and N. Leone. Enhancing DLV instantiator by backjumping techniques. *Ann. Math. Artif. Intell.*, 51(2-4):195–228, 2007.

[34] R. Pichler and V. Savenkov. DEMo: Data Exchange Modeling Tool. In *VLDB*, 2009.

[35] I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term Indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26. Elsevier Science, 2001.

[36] Stephan Schulz. System Description: E 1.8. In *LPAR*, 2013.

[37] SMT-LIB. `http://smtlib.cs.uiowa.edu/`.

[38] B. ten Cate, L. Chiticariu, P. Kolaitis, and W. C. Tan. Laconic Schema Mappings: Computing Core Universal Solutions by Means of SQL Queries. In *VLDB*, 2009.

[39] TPC. `http://www.tpc.org/`.

[40] TPTP. `http://www.cs.miami.edu/~tptp/`.