

Benchmarking the Chase

Michael Benedikt¹
Boris Motik¹

¹ University of Oxford
Oxford, UK

George Konstantinidis¹
Paolo Papotti³
Efthymia Tsamoura¹

² University of Basilicata
Potenza, Italy

Giansalvatore Mecca²
Donatello Santoro²

³ Arizona State University
Phoenix, AZ, USA

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, also known as *dependencies*, expressed as universal implications possibly containing existential quantification in the conclusion. When applied to a set of dependencies 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, but over the last decade practical aspects, such as developing optimizations of the chase algorithms and building chase-based systems for various tasks, have also been considered. Although algorithms for chasing source-to-target dependencies have shown promise in practice, with target dependencies scalability has been achieved only in quite restricted cases. The performance of chase implementations on large sets of complex dependencies and large datasets remains unknown. This suggests 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 techniques 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 developed dependency sets with different structural properties, and datasets of varying sizes.

We also 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 over 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?
- What 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?

In an attempt to answer these questions, we considered 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 re-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PODS'17, May 14–19, 2017, Chicago, Illinois, USA

© 2017 ACM. ISBN 978-1-4503-4198-1/17/05...\$15.00

DOI: <http://dx.doi.org/10.1145/3034786.3034796>

lated 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, \dots, t_n as \vec{t} , and we often treat it 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 all values occurring in a tuple of some $I(R)$. A *relational atom* has the form $R(\vec{t})$ where R is an n -ary relation and \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 $\sigma = \{t_1 \mapsto s_1, \dots, t_n \mapsto s_n\}$ to denote that $\sigma(t_i) = s_i$ for $1 \leq i \leq n$. 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 $\sigma(t)$ (i.e., terms outside the domain of σ remain unchanged). The *composition* of σ with a term mapping μ is the term mapping $\sigma \circ \mu$ whose domain is the union of the domains of σ and μ , and it is defined by $(\sigma \circ \mu)(t) = \sigma(\mu(t))$. 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 $\lambda(\vec{x})$ and $\rho(\vec{x}, \vec{y})$ are conjunctions of relational, null-free atoms whose free variables are contained in \vec{x} , and $\vec{x} \cup \vec{y}$ correspondingly.

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

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 $\{x_i, x_j\} \subseteq \vec{x}$.

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

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, and we commonly omit the leading universal quantifiers.

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 dependency 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 *trigger* for τ in I is a homomorphism h of $\lambda(\vec{x})$ into I . Moreover, an *active trigger* for τ in I is a trigger h for τ in I such that, if τ is a TGD, then no

extension of h to a homomorphism of $\rho(\vec{x}, \vec{y})$ into I exists, and if τ is an EGD, then $h(x_i) \neq h(x_j)$. Finally, dependency τ is satisfied in I if there does not exist an active trigger for τ in I .

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 $\tau \in \Sigma$. 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 $\mu(J) \subseteq J'$ and $\mu(c) = c$ holds for each constant value $c \in \text{Const}$. Solutions for Σ and I 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 $\sigma(x_1), \dots, \sigma(x_n)$. 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 [15].*

The chase. The chase modifies an instance by a sequence of *chase steps* until all dependencies are satisfied. Let I be an instance, let τ be a TGD or an EGD of the form (1) or (2), and let h be a trigger for τ in I . If τ is a TGD, applying the chase step for τ and h to I extends I with facts of the conjunction $h'(\rho(\vec{x}, \vec{y}))$, 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, if τ is an EGD, applying the chase step for τ and h to I *fails* if $h(x_i) \neq h(x_j)$ and $\{h(x_i), h(x_j)\} \subseteq \text{Const}$, and otherwise it computes $\mu(I)$ where $\mu = \{h(x_j) \mapsto h(x_i)\}$ if $h(x_i) \in \text{Const}$, and $\mu = \{h(x_i) \mapsto h(x_j)\}$ if $h(x_i) \notin \text{Const}$.

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, \dots 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 $i \geq 0$, 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 . Still, each result I_∞ of a chase sequence for Σ and I is a universal solution for Σ and I [15].

A finite chase sequence is *terminating*. 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 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 remove all substitutions that are not null-free. Checking if a set of dependencies Σ has terminating chase is undecidable [13]. *Weak acyclicity* [15] was the first suffi-

Algorithm 1 RESTRICTED-CHASE(Σ, I)

```
1:  $\Delta I := I$ 
2: while  $\Delta I \neq \emptyset$  do
3:    $N := \emptyset, \quad \mu := \emptyset$ 
4:   for each  $\tau \in \Sigma$  with body  $\lambda(\vec{x})$  do
5:     for each trigger  $h$  for  $\tau$  in  $I$  such that  $h(\lambda(\vec{x})) \cap \Delta I \neq \emptyset$  do
6:       if  $h$  is an active trigger for  $\tau$  in  $\mu(N \cup I)$  then
7:         if  $\tau = \forall \vec{x} \lambda(\vec{x}) \rightarrow \exists \vec{y} \rho(\vec{x}, \vec{y})$  is a TGD then
8:            $h' := h \cup \{\vec{y} \mapsto \vec{v}\}$  where  $\vec{v} \subseteq \text{Nulls}$  is fresh
9:            $N := N \cup h'(\rho(\vec{x}, \vec{y}))$ 
10:        else if  $\tau = \forall \vec{x} \lambda(\vec{x}) \rightarrow x_i = x_j$  is an EGD then
11:          if  $\{h(x_i) \neq h(x_j)\} \subseteq \text{Const}$  then fail
12:           $\omega := \{\max(h(x_i), h(x_j)) \mapsto \min(h(x_i), h(x_j))\}$ 
13:           $\mu := \mu \circ (\omega \circ \mu)$ 
14:    $\Delta I := \mu(N \cup I) \setminus I$ 
15:    $I := \mu(I) \cup \Delta I$ 
```

cient polynomial-time condition for checking if Σ has terminating chase. Stronger sufficient (not necessarily polynomial-time) conditions have been proposed subsequently [20, 32].

Data exchange. In relational-to-relational *data exchange* [15], 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 [15].

3. IMPLEMENTING THE CHASE

In this section we discuss the main challenges that chase implementations must address. Our discussion is largely independent of the specific technology, but we discuss briefly how these issues impact RDBMS-based implementations.

General structure. Computing the chase is naturally expressed as a fixpoint computation. Algorithm 1 achieves this by following the definition of a chase sequence from Section 2: in each iteration (lines 2–15), the algorithm examines each dependency τ (line 4) and trigger h (line 5), checks whether h is active (line 6), and, if so, applies the chase step (lines 8–9 for TGDs and lines 11–13 for EGDs). Although chase implementations may and do depart from the specifics of Algorithm 1, the algorithm is useful because it allows us to discuss the following issues, which each implementation must address in one form or another.

- To avoid an important source of inefficiency, we must ensure that each relevant trigger for τ and I is considered in line 5 at most once.
- Applying chase steps should not interfere with the enumeration of triggers.
- The EGD chase steps are complex and require care.
- Checking whether a trigger is active can be expensive.

Identifying triggers. Let τ be a TGD or an EGD of the form (1) or (2), respectively, and let I be a finite instance. Clearly, substitution h is a trigger for τ and I if and only if h is an answer to $\lambda(\vec{x})$ on I . Hence, the triggers for τ can be determined in line 5 by evaluating $\lambda(\vec{x})$ as a CQ over I . The latter can be solved using any known join algorithm, and an RDBMS-based implementation

can translate $\lambda(\vec{x})$ into SQL in a straightforward manner. However, evaluating $\lambda(\vec{x})$ in each iteration “from scratch” would be very inefficient since it would repeatedly consider triggers from all previous iterations. Algorithm 1 thus uses *seminaïve* evaluation [1]. It maintains an auxiliary set ΔI of “newly derived facts” from the most recent iteration, and it requires at least one atom of $h(\lambda)$ in line 5 to be contained in ΔI . As a consequence, each combination of τ and h is considered at most once during algorithm’s execution.

Applying chase steps. An important consideration is how query $\lambda(x)$ is evaluated over I in line 5: we must ensure that we do not miss any applicable triggers, and that we retrieve each such trigger once. One possibility is to compute and store the result in a temporary relation, but this can impose a significant overhead. It is therefore often preferable to evaluate $\lambda(x)$ over I in a “streaming” mode, where a trigger h is returned as soon as it is computed; but then, set I should not change in lines 4–13 or the modifications to I could affect the enumeration of triggers in line 5. To ensure this, the chase steps in Algorithm 1 do not modify I directly. Instead, all changes are accumulated in an auxiliary set N and a term mapping μ ; after each iteration, ΔI contains the “newly derived subset” of N (line 14), which is propagated to I (line 15). The algorithm terminates when ΔI becomes empty in line 2 since another iteration then cannot derive new facts.

Handling EGDs. Applying a chase step to an EGD τ of the form (2) and trigger h poses several problems. A minor issue is that, when $h(x_i)$ and $h(x_j)$ are both labeled nulls, we can replace either value with the other. To make this choice deterministic, one 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 also ensures uniqueness of the chase for a specific chase variant that we discuss shortly.

A more complex issue is to ensure that I does not change in lines 5–13. To achieve this, Algorithm 1 uses a term mapping μ that accumulates all the required changes to I (line 13): at the end of each iteration, $\mu(v)$ is defined for each labeled null v that is to be replaced with $\mu(v)$. The expression in line 13 ensures that μ is correctly updated when several EGD steps are applied in single iteration. For example, consider $\mu = \{v_1 \mapsto v_2, v_3 \mapsto v_4\}$ and $\omega = \{v_2 \mapsto v_3\}$; then, $\omega \circ \mu = \{v_1 \mapsto v_3, v_2 \mapsto v_3, v_3 \mapsto v_4\}$ “normalizes” μ with ω , and $\mu \circ (\omega \circ \mu) = \{v_1 \mapsto v_4, v_2 \mapsto v_4, v_3 \mapsto v_4\}$ reflects the cumulative effects of all EGDs.

At the end of each iteration (lines 14–15), the facts in both I and N may require updating, and all facts newly derived in the iteration must be added to ΔI to correctly further trigger dependencies.

Checking active triggers. Assume that h is a trigger for a dependency τ in an instance I , and consider checking whether h is an active trigger for τ in $\mu(N \cup I)$ in line 6. It should be clear that h is a trigger for τ in $\mu(N \cup I)$ if and only if $\mu(h(x)) = h(x)$ for each variable x from the domain of h . In other words, we simply check whether “ h is up-to-date with μ .” Thus, to complete the active trigger check, if τ is of the form (2), we check whether $h(x_i) \neq h(x_j)$ holds. If τ is of the form (1), we check whether $\gamma(\mu(I \cup N)) = \emptyset$ where $\gamma = \exists \vec{y} \rho(h(\vec{x}), \vec{y})$ is a boolean CQ; provided that $\mu(I \cup N)$ is explicitly available, this condition can be checked in an RDBMS-based system by simply extending the CQ from line 5 with a NOT EXISTS (SELECT * WHERE ...) condition. In practice, however, materialising $\mu(I \cup N)$ after each update is typically very expensive, which introduces a major source of difficulty that can be addressed in several different ways. If an exact active trigger check is desired, one can abandon the seminaïve evaluation in Algorithm 1 and replace lines 5–13 with a loop that selects just one active trigger

for one dependency and applies the chase step immediately; however, as we discuss in Section 7, this usually results in very poor performance. Another option is to relax the active trigger check and use a sound but incomplete test; we discuss several such options in the rest of this section, each giving rise to a different chase variant. Algorithm 1 still produces a universal solution in such cases, but it may apply more chase steps than with the exact active trigger check in place.

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

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

$$R(x_1, x_2) \rightarrow \exists y R(x_1, y) \wedge A(y) \wedge A(x_2) \quad (3)$$

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

Dependencies in Σ are weakly acyclic [15] so the restricted chase terminates on all finite instances, but the chase 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 $A(b)$. In contrast, 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 prior to applying the chase: for each TGD $\tau \in \Sigma$ of the form (1) such that $\rho(\vec{x}, \vec{y})$ 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.

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

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

In addition to the difficulties of checking whether trigger h is active we outlined earlier, the dependence on the ordering of chase steps is problematic since it can make the chase more difficult to analyze from a theoretical point of view. This motivates chase variants where the check in line 6 is either eliminated or weakened.

The *unrestricted* (or *oblivious*) chase simply 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, the 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.

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

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 [20] 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 the set of facts I from Example 1 produces facts $R(a, f(a))$, $A(f(a))$, $A(b)$, $R(b, f(b))$, 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* [13] weakens line 6 so that it checks whether h is active in I , rather than in $\mu(N \cup I)$; since I is fixed in an iteration, this can make checking active triggers much easier to implement. Known acyclicity conditions [20] 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).

While 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 two chase variants never produces more facts than the original variant, and in certain cases it can produce fewer facts.

The *frugal chase* [22] further considers triggers that are only partially active. Let τ be a TGD τ of the form (1) such that $\rho(\vec{x}, \vec{y})$ can be rewritten as $\rho_1(\vec{x}_1, \vec{y}_1) \wedge \rho_2(\vec{x}_2, \vec{y}_2)$ where \vec{x}_1 and \vec{x}_2 , and \vec{y}_1 and \vec{y}_2 are not necessarily disjoint, and let h be a trigger for τ in an instance I . Then, h is *partially active* for τ in I if h can be extended to a homomorphism h' such that (i) $\rho_1(h'(\vec{x}_1), h'(\vec{y}_1)) \subseteq I$, (ii) $h'(\vec{y}_1)$ contains only labeled nulls, and (iii) for each fact $P(\vec{w}) \in I$ such that $\vec{w} \cap h'(\vec{y}_1) \neq \emptyset$, we have $P(\vec{w}) \in \rho_1(h'(\vec{x}_1), h'(\vec{y}_1))$ (i.e., each atom in I that joins “existentially” with the image of ρ_1 is contained in the image of ρ_1). Since ρ_1 is already satisfied, the frugal chase satisfies the TGD τ by extending I with $\rho_2(h''(\vec{x}_2), h''(\vec{y}_2))$, where h'' is a homomorphism that extends (one such) h' by mapping the variables in $\vec{y}_2 \setminus \vec{y}_1$ to fresh labeled nulls.

EXAMPLE 6. Let Σ contain the TGD (5) from Example 2 and TGD (11), and let I be as in (12).

$$B(x) \rightarrow \exists y R(x, y) \quad (11)$$

$$I = \{ B(a) \} \quad (12)$$

Assume now that the frugal chase first applies (11) and thus extends I with fact $R(a, v_1)$. Then, $h = \{x_1 \mapsto a, x_2 \mapsto v_1\}$ is a partially active trigger for TGD (5): we can decompose the head of (5) such that $\rho_1 = R(x_1, y)$ and $\rho_2 = A(y)$, and we can extend h to $h' = \{x_1 \mapsto a, x_2 \mapsto v_1, y \mapsto v_1\}$ such that $h'(\rho_1)$ is contained in the current instance, and $h'(y) = v_1$ does not occur in the instance outside of $h'(\rho_1)$. Hence, the frugal chase satisfies (5) by adding $A(v_1)$ to the instance, instead of introducing a fresh labelled null.

The core chase [13] omits the active trigger check completely; however, after each iteration it replaces the current instance with its core [16]—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 objectives are to determine whether existing chase implementations can support data exchange and query answering on nontrivial inputs, and to identify the implementation decisions most relevant to performance. To answer these questions, we used nine publicly available systems shown 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 [19]. The system runs on top of PostgreSQL or HSQLDB, and we used the former in our experiments.

CHASEFUN [12] 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 that orders TGD and EGD chase steps 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 [18] was initially developed for data cleaning, but has since been redesigned as an open-source data exchange system. It handles s-t TGDs and target TGDs and EGDs, and it can also compute certain query answers over the target instance. 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. Three of our test systems 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 [14], which internally uses the restricted chase implemented in RAM.

PDQ [10, 11] 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, PDQ reduces query reformulation to checking query containment under TGDs and/or EGDs, and the latter problem is solved using an implementation of the restricted chase on top of PostgreSQL.

Systems motivated by query answering. GRAAL [7] is an open-source toolkit 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. GRAAL can be used both in an RAM and on secondary storage, such as an RDBMS, a triple store, or a graph database. We used the RAM-based version as we found it to be faster.

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 very 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 [24] is a mature disjunctive Datalog system supporting a range of features such as 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 [23], 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 obtain certain answers, we externally postprocess 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 over the RDF data model and without the unique name assumption. 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 \mathcal{F} and a conjecture H , and it decides the unsatisfiability of $\bigwedge \mathcal{F} \wedge \neg H$. 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 \mathcal{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, \mathcal{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 \mathcal{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 $\bigwedge \mathcal{F} \wedge \neg \sigma(H)$ 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 \mathcal{F} encodes dependencies on

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 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 consists of two distinct parts. The first part is described 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 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 IBENCH [4], as well as instances of varying sizes produced by the instance-generation tool TOXGENE [8]. We divide our scenarios into the following two groups.

Correctness scenarios were designed to verify that the systems correctly produce universal solutions. Checking homomorphisms between solutions is computationally challenging so these scenarios 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 involves choices that can significantly 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 such formats are difficult to read for humans and so we found them inappropriate for CHASEBENCH. 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 com-

mon 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 parse and 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 initially produced by TOXGENE 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, it 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 remain 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. In our experiments we used the generator to develop scenarios from scratch, but the generator can also be used to increase the difficulty of existing scenarios.

Our generator is based on our earlier approach [22] for generating linear dependencies, which we extended to support an arbitrary number of body atoms. The generator creates a predicate name space of a certain required size, and it uniformly chooses a (parameterized) number of predicates to populate a conjunction of atoms (which can become a query or a dependency). A parameter is used to control the maximum number of repeated relations in this formula, and another parameter is used to determine the arity of the atoms. The generator can create “chain” conjunctions (the last variable of an atom is joined with the first one of the next atom), “star” conjunctions (the first variable of all atoms also occurs in a designated “center” atom that has a different arity than the other atoms), and “random” ones in which variables are chosen out of a variable

name space and are distributed in a conjunction (the larger the variable name space, the fewer joins are introduced). Each dependency is generated by first creating a conjunction and then selecting the subset of atoms that make up the head. The size of the dependencies’ bodies can be fixed; for example, this parameter was 1 in our DEEP scenarios, thus producing linear TGDs. The generator maintains a weakly-acyclic dependency graph and tests each generated dependency against the ones there were created previous; if the newly created dependency violates weak acyclicity, it is discarded and a new dependency is produced.

4. The query generator. To evaluate the performance of computing certain answers, nontrivial queries over the target schema are required. Existing benchmarks such as LUBM [21] come with several manually curated queries, which we incorporated into our scenarios. To obtain queries for scenarios where the target schema is automatically generated, we developed a new query generator. The tool is given a desired number of joins in the output query N_J , the desired number of selections conditions in the output query N_S , the percentage of attributes that will be projected P , and a set of tables of the target schema. It first identifies a set of joinable attribute pairs from the tables of the source and target schema by analyzing the dependencies. Two attributes are considered joinable if they join either in the head or the body of the rule; moreover, if a variable is passed from the body to the head of the rule, then the attributes that map to these variables are considered joinable as well; and finally, the joinable attributes are transitively closed. Next, the algorithm iteratively creates N_J join conditions by choosing at random a table T and attribute a from the schema and then choosing a random table and attribute that is joinable with $T.a$. The algorithm then proceeds analogously to create N_S selections by first choosing an attribute that does not appear in a join condition and then introducing randomly selected value from the attribute’s domain. Finally, the algorithm randomly projects P attributes. By focusing on joinable attributes, the algorithm is more likely to produce a query with a nonempty output, but does not guarantee it; therefore, we discarded and regenerated each query that produced an empty output on the target instance as long as needed.

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 into five families, as 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 identify each scenario using a unique identifier; for DOCTORS and LUBM scenarios, the identifier is obtained by combining the scenario name with the source instance size, such as LUBM-90k. We discuss next the main features of our scenario families.

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 scenarios from this family are based on the DOCTORS data integration task from the schema mapping literature [17]. 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 10k, 100k, 500k, 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 [21] 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 queries designed to test various aspects of query answering. Using the LUBM data generator we produced instances with 90k, 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 source instance using 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 $p_{src}(s, o)$ 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 consequence 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

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

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, thus obtaining two scenarios of the IBENCH family.

- STB-128, derived from an earlier ST-benchmark [3], is the smaller scenario of the family.
- ONT-256, a scenario motivated by ontologies, has several times larger source instance.

For both of these scenarios, we used the integration of IBENCH and TOXGENE to generate 1k 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 $x \in \text{Vars}$ into a distinct constant value $\sigma(x) \in \text{Const}$; 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 500M facts on the largest DEEP300 scenario.

The TGDs were taken from previous work [22] 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 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. All systems eventually passed all correctness tests, except for PE-

GASUS which failed two correctness tests but could not be updated. Complete results of the performance tests, including a breakdown of all times, are given in the appendix 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 .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 all of these tasks; 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 supported such time measurements. 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 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 the appendix. 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 do not report any 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

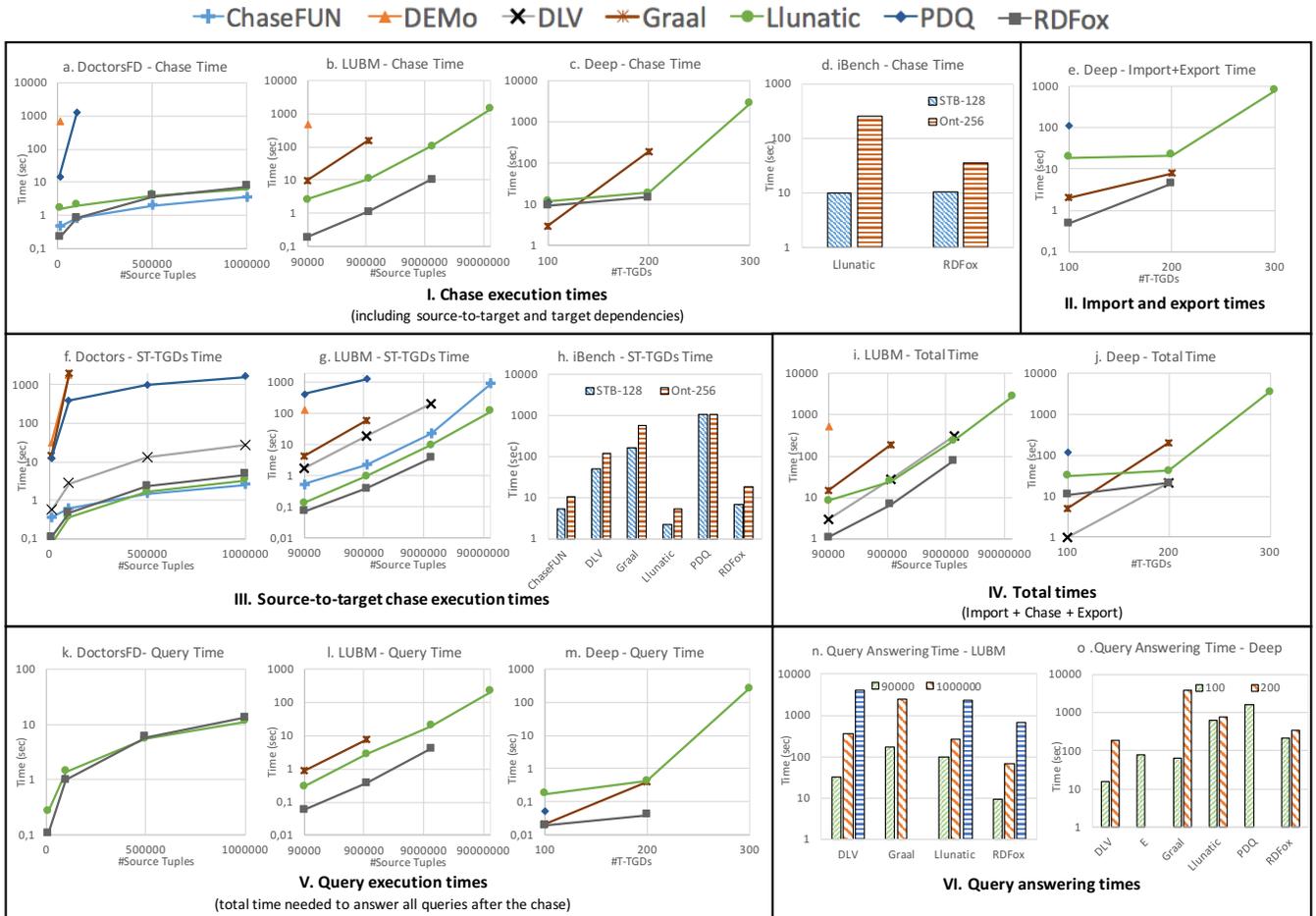


Figure 1: Experimental results for scenarios where times vary significantly between system configurations

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” test 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” test protocol, 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 features, 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 tested 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, 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 exam-

ple, PDQ completed only five out of the 17 data exchange tests. CHASEFUN completed all tests that it was applicable to.

7. LESSONS LEARNED

Our tests 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 the appendix shows all results), we obtained several important insights. Interestingly, the tradeoffs associated with this question are quite different for RDBMS- and RAM-based systems.

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

Implementing the restricted chase in RDBMS-based systems is quite challenging. Triggers are retrieved in such systems using SQL queries, so it is natural to embed the active trigger check into these queries. For example, query (13) 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
   WHERE R.a=R1.a AND R1.b=A1.a AND R.b=A2.a)

```

(13)

RDBMSs, however, evaluate queries fully before any updates, so query (13) 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 (13) and run one query per trigger, but this is very slow on large instances in most RDBMSs. Table 7 confirms this: with both the 1-parallel 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 (13), 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 (13), 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 the appendix 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 the appendix. On the 10k and 100k scenarios, the core target instances were 18% smaller than the ones produced using unrestricted Skolem chase, suggesting that scalable methods for core computation in a more general setting could be practically relevant.

Implementing EGDs. In order to apply the EGDs, a system typically first retrieves and deletes all affected facts, applies μ , and inserts 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, which is often string-based; for example, string values are encoded in query (10) using the '_5k_' 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 nulls; 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 attribute of the target relation: 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 all relevant queries. Most queries can be answered very quickly (typically in under 1 s). Query evaluation was slowest on DEEP300 and LUBM-120M because the target instances of that scenario are much larger than in the case of other scenarios.

We analyzed the queries in our benchmark to detect a possible correlation between execution times and various complexity parameters, such as the number of variables in the head and the body, the number of joins and so on; the parameters for all queries are shown on the benchmark Web site. 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 using the following steps:

- 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, data access structures such as indexes are smaller so joins are faster. Fourth, dictionary encoding removes a problem specific to the Skolem chase: target TGDs

may produce very “deep” Skolem terms that can be expensive to manage, but dictionary encoding reduces the depth of such terms.

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. First, we measured the chase time on the unencoded instance. Second, 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. In our performance comparison, LLUNATIC was configured to turn on the encoding on all scenarios with target TGDs.

Chase vs. first-order theorem proving. E 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 given in the appendix): 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 (i.e., “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 obtained from the queries being reformulated, rather than for 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 experiments also gave us some insight regarding directions for future work in the community.

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 hypothe-

ses 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.

Understanding the chase with EGDs. The handling of EGDs in the chase is an area in need of much further study, both in terms of theory (e.g., termination guarantees and static analysis) and implementation. Equality reasoning has been studied extensively in the theorem-proving community (e.g., [6]), and thus a first step would be to better understand the applicability of these techniques to the chase with EGDs.

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.

Alternative approaches to query answering. The chase is often used 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. 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. The classical chase procedure is particularly well-suited for use cases where the certain answers for many queries must be computed for the same instance. For comparison with other approaches, it might be appropriate to consider a more dynamic setting, as well as possibly develop a set of complexity measures appropriate to such a setting.

Answering queries without materializing the chase. As our experiments show, the chase can be very large so computing and maintaining it can be impractical. It can sometimes thus be desirable to answer queries without first materializing the chase. A lot of work has already been invested in *query rewriting*, where a query is transformed to take into account the effect of dependencies so that the transformed query can be evaluated directly over the source instance. This, however, is feasible only with simple dependencies such as linear TGDs, and answering queries in cases where query rewriting is not applicable remains an open problem. Goal-oriented query answering techniques from Datalog such as *magic sets* [9] provide a promising starting point for this investigation.

Ordering of steps. 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. More research is needed to understand the impact of the ordering: even with TGDs only, a particular order can make the active triggers check more effective.

9. RELATED WORK

Experimental evaluations of many chase based systems have already been conducted [34, 28, 27, 17, 22, 12]. 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 tested systems. We evaluate the systems on over 20 scenarios covering a wide range of assumptions, testing both correctness and scalability, covering a broad 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. 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 our work can be easily extended to support new classes of dependencies and other chase-related tasks, such as 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. ACKNOWLEDGMENTS

The work by Benedikt, Motik, and Konstantinidis was funded by the EPSRC grants PDQ (EP/M005852/1), ED³ (EP/N014359/1), DBOnto (EP/L012138/1), and MaSi³ (EP/K00607X/1).

12. 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] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [7] 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.
- [8] D. Barbosa, A. Mendelzon, and K. Keenleyside, J. and Lyons. ToXgene: A template-based data generator for XML. In *SIGMOD*, 2002.
- [9] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS*, pages 269–283, 1987.
- [10] M. Benedikt, J. Leblay, and E. Tsamoura. PDQ: Proof-driven query answering over web-based data. In *VLDB*, 2014.
- [11] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. In *VLDB*, 2015.
- [12] A. Bonifati, I. Ileana, and M. Linardi. Functional Dependencies Unleashed for Scalable Data Exchange. In *SSDBM*, 2016.
- [13] A. Deutsch, A. Nash, and J. Remmel. The chase revisited. In *PODS*, 2008.
- [14] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.
- [15] R. Fagin, P.G. Kolaitis, R.J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.
- [16] R. Fagin, P.G. Kolaitis, and L. Popa. Data Exchange: Getting to the Core. *TODS*, 30(1):174–210, 2005.
- [17] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, 2014.
- [18] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That’s All Folks! LLUNATIC Goes Open Source. In *VLDB*, 2014.
- [19] G. Gottlob and A. Nash. Efficient Core Computation in Data Exchange. *J. of the ACM*, 55(2):1–49, 2008.
- [20] 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.
- [21] 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.
- [22] G. Konstantinidis and J. L. Ambite. Optimizing the chase: Scalable data integration under constraints. In *VLDB*, 2015.
- [23] N. Leone, M. Manna, G. Terracina, and P. Veltri. Efficiently computable Datalog[∃] programs. In *KR*, 2012.
- [24] 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.
- [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 *Handbook of Automated Reasoning*, volume I. Elsevier, 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. Art. Int.*, 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 *Handbook of Automated Reasoning*. Elsevier, 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/>.

APPENDIX

A. FULL EXPERIMENTAL RESULTS

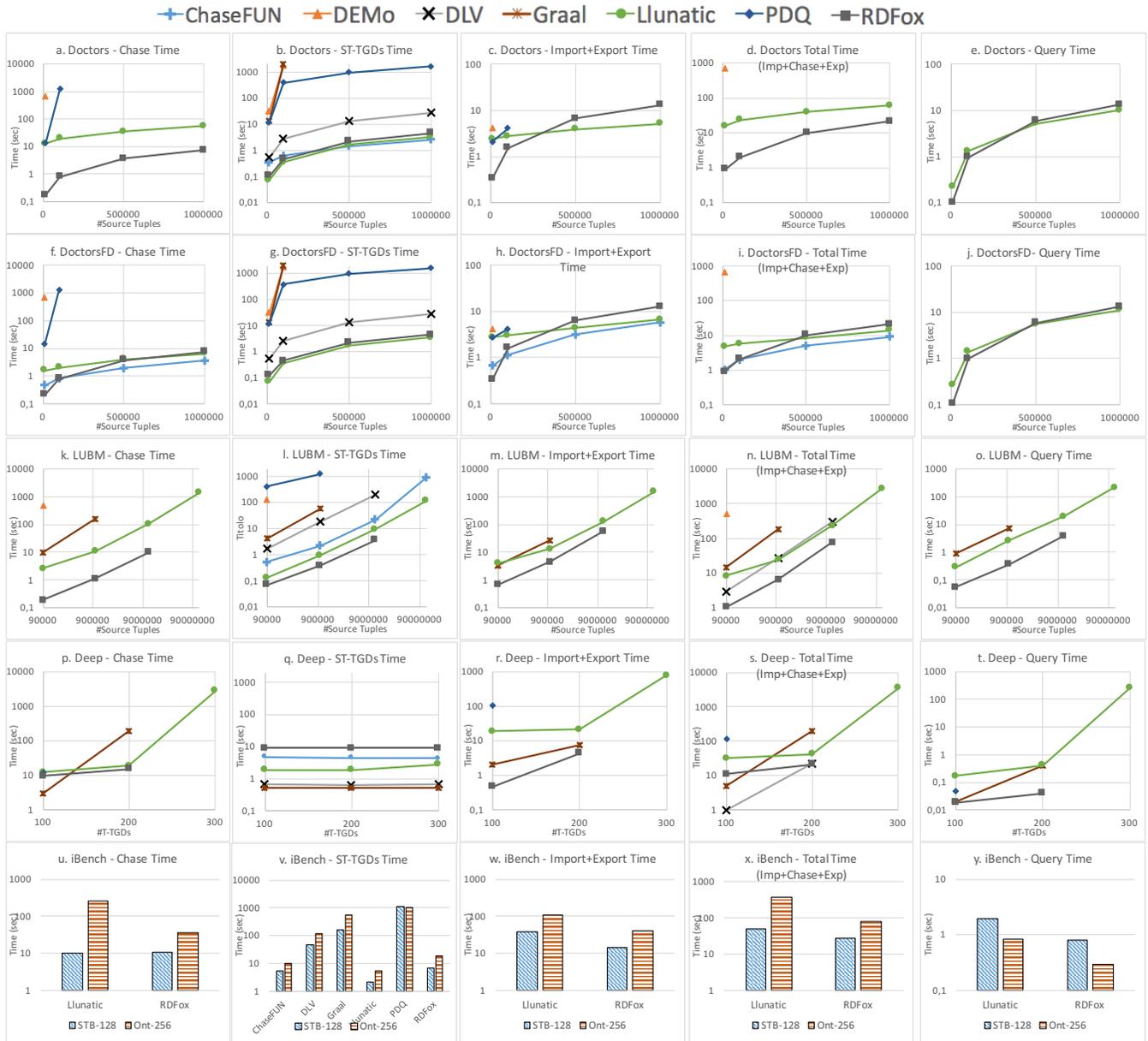


Figure 2: Full charts of the chase experiments

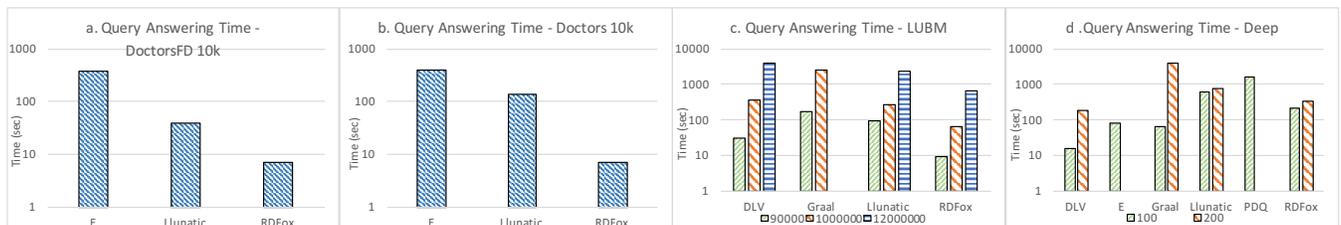


Figure 3: Full charts of the query answering experiments

Chase Times																				
Doctors																				
10k					100k					500k					1m					
Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	
ChaseFUN	10000	10000	10000	10000	100000	100000	100000	100000	100000	500000	500000	500000	500000	500000	1000000	1000000	1000000	1000000	1000000	
DEMO	686.00	0.35	4.14	691.00		0.62					1.49					2.58				
DLV		0.54				1777.00										13.06				
Graal		13.71				1936.96										26.98				
Llunatic	12.63	0.08	2.37	15.78	0.22	19.56	0.35	2.75	23.70	1.30	34.76	1.64	3.86	39.82	5.18	55.96	3.32	5.11	62.00	10.00
PDQ	13.55	12.00	2.06			1220.00	388.00	4.06							977.00				1622.00	
RDFox	0.19	0.11	0.33	0.90	0.10	0.81	0.46	1.53	2.03	0.97	3.73	2.26	6.57	10.12	5.88	7.53	4.57	13.01	21.59	13.41

DoctorsFD																				
10k					100k					500k					1m					
Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	
ChaseFUN	10000	10000	10000	10000	100000	100000	100000	100000	100000	500000	500000	500000	500000	500000	1000000	1000000	1000000	1000000	1000000	
DEMO	658.00	0.46	0.66	1.00	0.83		1.14	2.00			1.97		3.17	5.00		3.55		5.75	9.00	
DLV		0.56				1777.00										13.03				
Graal		13.71				1936.96										27.05				
Llunatic	1.60	0.07	2.73	4.74	0.26	2.02	0.37	3.04	5.59	1.41	3.99	1.77	4.32	8.46	5.54	6.48	3.52	6.64	13.78	11.22
PDQ	14.50	12.00	2.66			1258.00	388.00	4.14							977.00				1622.00	
RDFox	0.21	0.13	0.33	0.90	0.10	0.81	0.46	1.60	2.03	0.97	3.73	2.27	6.39	10.15	5.85	7.49	4.56	12.67	20.77	13.23

LUBM																				
001					010					100					01k					
Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	
ChaseFUN	90000	90000	90000	90000	1000000	1000000	1000000	1000000	1000000	12000000	12000000	12000000	12000000	12000000	120000000	120000000	120000000	120000000	120000000	
DEMO	503.00	0.54	3.44	512.00		2.29					22.17					906.51				
DLV		1.78		3.00		19.04			28.00		197.10		304.00							
Graal	9.73	4.44	3.37	14.90	0.90	162.00	59.02	27.00	188.56	7.44										
Llunatic	2.67	0.13	4.03	8.29	0.29	10.79	0.95	13.13	24.31	2.69	107.35	9.47	129.21	237.44	19.56	1415.53	117.82	1543.17	2739.29	214.71
PDQ		410.00				1276.00														
RDFox	0.19	0.07	0.68	1.06	0.06	1.12	0.39	4.59	6.63	0.37	10.34	3.65	57.03	77.04	3.96					

iBench										
STB-128					Ont-256					
Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	
ChaseFUN	STB-128	STB-128	STB-128	STB-128	Ont-256	Ont-256	Ont-256	Ont-256	Ont-256	
DLV	5.46				10.23					
Graal	48.59				118.01					
Llunatic	160.38				566.46					
PDQ	10.10	2.16	39.31	50.08	1.92	257.12	5.34	108.55	366.18	0.82
RDFox	1088.00				1045.00					
RDFox	10.73	6.65	14.22	27.19	0.81	35.27	18.73	40.87	82.71	0.29

Deep															
100.00					200.00					300.00					
Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	Chase	Chase ST-TGDs	Import/Exp	Total time	Query	
ChaseFUN	100	100	100	100	200	200	200	200	200	300	300	300	300	300	
DLV	4.73				4.50					4.31					
Graal	0.67		1.00		0.65			22.00		0.67					
Llunatic	2.98	0.53	2.00	4.98	0.02	192.00	0.52	7.61	199.61	0.39	0.52				
PDQ	12.13	1.90	18.96	31.83	0.17	19.25	1.86	21.51	41.59	0.41	2805.48	2.79	770.15	3576.85	253.16
RDFox	11.82		106.51	118.33	0.05										
RDFox	9.49	8.93	0.47	10.98	0.02	15.23	9.03	4.41	20.96	0.04	8.86				

Unsupported
Timeout

Table 5: Full results of the chase experiments

Query Answering Times		
	Doctors	DoctorsFD
	10k	10k
E	399.00	383.00
Llunatic	138.05	38.63
RDFox	6.96	7.02

LUBM			
	001	010	100
DLV	31.00	356.00	3960.00
Graal	176.32	2479.28	
Llunatic	95.42	267.31	2393.79
RDFox	9.26	66.47	653.21

Deep		
	100	200
DLV	16.00	184.00
E	79.00	
Graal	62.82	3843.99
Llunatic	609.39	783.85
PDQ	1566.79	
RDFox	212.64	333.78

Table 6: Full results of the query answering experiments

Doctors									
	10k			100k			500k		
	Chase Time	#Target Tuples	Query Time	Chase Time	#Target Tuples	Query Time	Chase Time	#Target Tuples	Query Time
Llunatic - 1-Parallel	12,63	9.734	0,22	19,56	81.000	1,30	34,76	397.000	5,18
Llunatic - Unrestricted	15,40	9.734	0,21	20,69	81.000	1,38	44,91	397.000	5,60
Llunatic - Restricted	69,12	9.734	0,21	3948,14	81.000	2,83			
RDfox - Unrestricted	0,18	9.734	0,10	0,81	81.000	0,97	3,73	397.000	5,88
RDfox - Restricted	0,14	9.734	0,11	0,67	81.000	0,98	2,92	397.000	5,75

LUBM						
	001			010		
	Chase Time	#Target Tuples	Query Time	Chase Time	#Target Tuples	Query Time
Llunatic - 1-Parallel	2,67	141.213	0,29	10,79	1.779.697	2,69
Llunatic - Unrestricted	6,37	177.738	0,31	32,99	2.246.699	1,69
Llunatic - Restricted	2196,80	141.213	0,23			
RDfox - Unrestricted	0,19	177.738	0,06	1,12	2.246.699	0,45
RDfox - Restricted	0,21	141.213	0,06	0,94	1.779.697	0,42

iBench						
	STB-128			Ont-256		
	Chase Time	#Target Tuples	Query Time	Chase Time	#Target Tuples	Query Time
Llunatic - 1-Parallel	10,10	1.918.178	1,92	257,12	5.673.830	0,82
Llunatic - Unrestricted	34,04	1.918.217	1,40	307,13	5.674.103	0,73
Llunatic - Restricted						
RDfox - Unrestricted	10,73	1.918.217	0,81	35,27	5.674.103	0,29
RDfox - Restricted	10,87	1.918.178	0,78	36,83	5.673.830	0,30

Deep						
	100			200		
	Chase Time	#Target Tuples	Query Time	Chase Time	#Target Tuples	Query Time
Llunatic - 1-Parallel	12,13	18.386	0,17	19,25	902.636	0,41
Llunatic - Unrestricted	15,89	19.537	0,22	33,35	926.324	0,44
Llunatic - Restricted	22,23	18.347	0,17	7521,92	893.990	0,36
RDfox - Unrestricted	9,49	19.537	0,02	15,23	926.324	0,03
RDfox - Restricted	18,86	18.385	0,02	24,02	892.516	0,03

Table 7: Impact of the chase variants

Doctors (S-T tgds only)										
	Unrestricted		Restricted Less Favorable		Restricted Random		Restricted Favorable		Core	
	Chase Time	# Tuples	Chase Time	# Tuples	Chase Time	# Tuples	Chase Time	# Tuples	Chase Time	# Tuples
10k	0,17	11.808	79,40	10.208	89,33	9.867	87,96	9.734	0,25	9.734
100k	1,14	97.500	12404,00	95.634	11340,00	85.168	103720,00	81.000	0,73	81.000

Table 8: Impact of the core